Automatische Optimierung und Evaluierung modellbasierter Testfälle für den Komponenten- und Integrationstest

Der Technischen Fakultät der

Universität Erlangen-Nürnberg

zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Florin-Avram Pinte

Erlangen - 2012

Als Dissertation genehmigt von

der Technischen Fakultät der

Universität Erlangen-Nürnberg

Tag der Einreichung:	28.11.2011
Tag der Promotion:	27.04.2012
Dekan:	Prof. Dr. Marion Merklein
Berichterstatter:	Prof. Dr. Francesca Saglietti
	Prof. Dr. Fevzi Belli

Kurzfassung

In Rahmen dieser Arbeit wird ein werkzeuggestütztes Verfahren vorgestellt, das zum einen die automatische Testfallgenerierung anhand von UML-Modellen ermöglicht und zum anderen die generierten Testfallmengen hinsichtlich ihres Fehlererkennungspotentials zu bewerten erlaubt.

Das Testfallgenerierungsverfahren unterstützt den modellbasierten Komponenten- und Integrationstest. Zu den betrachteten Überdeckungskriterien zählen sowohl etablierte Überdeckungskriterien für den Komponententest als auch eine Reihe so genannter zustandsbasierter Integrationstestkriterien. Durch Benutzung genetischer Algorithmen ermöglicht das entwickelte Werkzeug optimierte Testfallmengen automatisch zu generieren, die nicht nur eine hohe Überdeckung bezüglich der betrachteten Kriterien erreichen, sondern auch eine möglichst geringe Anzahl an Testfällen enthalten. Des Weiteren unterstützt das Werkzeug die Visualisierung der von generierten Testfällen erzielten Überdeckung auf dem Modell und den modellbasierten Regressionstest.

Zur Bewertung des Fehlererkennungspotentials werden sowohl Testfallmengen für den Komponententest als auch Testfallmengen für den Integrationstest betrachtet. Einerseits wird evaluiert, inwiefern solche Testfallmengen die Entdeckung von Modellfehlern erlauben. Dies wird durch ein Mutationstestverfahren auf Modellebene umgesetzt, das in einem ersten Schritt mehrere Mutanten des Modells automatisch erzeugt. Daraufhin wird durch Ausführung der Testfallmenge auf den Mutanten und auf dem initialen Modell der Anteil an erkannten Mutanten ermittelt. Andererseits werden modellbasierte Testfälle mit manuell erstellten Testfällen verglichen und das Potential zur Erkennung von Implementierungsfehlern evaluiert.

Abstract

This thesis presents a tool-supported approach which enables the automatic test case generation from UML models and allows evaluating the fault detection potential of the generated test cases.

The test case generation method supports both the model-based component and the integration testing phase. The considered coverage criteria include established coverage criteria for component testing as well as a series of so-called state-based integration testing criteria. By using genetic algorithms the developed tool automatically generates optimized test cases, maximizing the coverage achieved with regard to considered coverage criteria and minimizing the number of test cases. In addition, coverage visualization and model-based regression testing are also supported.

For the purpose of evaluating the fault detection capability both component and integration test cases are considered. On the one hand, mutation testing is used for evaluating to which extent such test cases allow the detection of modeling faults. To do so, as a first step several mutants of the model are automatically generated; successively the proportion of mutants detected is determined by executing the test cases on the mutants and on the initial model. On the other hand, the generated test cases were compared to manually created test cases in order to evaluate their potential for detecting implementation faults.

Danksagung

Diese Arbeit entstand am Lehrstuhl für Software Engineering im Rahmen des Forschungsprojekts UnITeD, das vom Bayrischen Wirtschaftsministerium gefördert wurde. An dieser Stelle möchte ich mich bei Frau Prof. Dr. Francesca Saglietti für die fachliche Betreuung und die Möglichkeit bedanken, an ihrem Lehrstuhl diese Promotion durchzuführen. Des Weiteren möchte ich mich bei Herrn Prof. Dr. Fevzi Belli, Herrn Prof. Dr. Klaus Meyer-Wegener und Herrn Prof. Dr. Bernhard Schmauss für die Beteiligung an meinem Promotionsverfahren bedanken.

Mein Dank gebührt ebenfalls meinen Kollegen Marc, Matthias, Sven, Raimar und meinen ehemaligen Kollegen Norbert, Dirk, Gerd, Josef, Jutta, Andrea für die tolle Zusammenarbeit und die schönen gemeinsamen Erlebnisse. In diesem Zusammenhang gilt mein Dank auch den Studien- und Diplomarbeitern, die ich betreuen durfte. Bedanken möchte ich mich auch bei der Firma Afra GmbH, insbesondere bei Herrn Mirco Richter und bei Herrn Gerhard Baier.

Mein tiefster Dank gilt meiner Familie und meinem Freundeskreis sowie ganz besonders meiner Freundin Elisabeth, die mich in allen Lebenslagen unterstützt haben.

Inhaltsverzeichnis

1. Einleitung

Heutzutage gibt es kaum noch Lebensbereiche, die ohne Unterstützung durch Softwaresysteme auskommen. Dies führt unter anderem dazu, dass Softwaresysteme immer höheren Qualitätsanforderungen genügen müssen. Besonders in sicherheitskritischen Bereichen wie z. B. der Medizin oder der Luftfahrt muss die eingesetzte Software *verlässlich* funktionieren. Darüber hinaus werden Softwaresysteme immer *komplexer*, wobei gleichzeitig ein immer höherer Zeit- und Kostendruck vorherrschen. All diese Aspekte führen dazu, dass die Erstellung solcher Systeme zu einer immer größeren Herausforderung wird.

Um diese Herausforderung meistern zu können, wurde Ende der 60er Jahre die Forderung laut, bei der Entwicklung von Software nach *ingenieurmäßigen Prinzipien* vorzugehen. Es entstand somit eine neue Disziplin mit dem Namen *Software Engineering*.

Mit Entstehung dieser Disziplin wurden Grundprinzipien der klassischen Ingenieurwissenschaften für die Softwareentwicklung übernommen. So hält z. B. das Prinzip der *Abstraktion* – als Hilfsmittel zur Bewältigung der Komplexität – mit der *modellbasierten Softwareentwicklung* auch in der Softwareindustrie Einzug. Ebenfalls zum Zweck der Bewältigung komplexer Problemstellungen eignet sich die *Teile und Herrsche*-Strategie, auf welcher das weit verbreitete *komponentenbasierte Softwareentwicklungsparadigma* beruht.

1.1. Motivation

Die vorhin genannten Entwicklungsparadigmen haben dazu geführt, dass sich die Art, wie Software entwickelt wird, bedeutend gewandelt hat. Heutzutage wird die zu erstellende Software nicht mehr als ein monolithisches System von Grund auf neu entwickelt. Vielmehr wird die gewünschte Systemfunktionalität in einzelne Teilfunktionalitäten aufgeteilt, um daraufhin für die Umsetzung der Teilfunktionalitäten entweder bestehende Komponenten wiederzuverwenden

oder neue Komponenten zu entwickeln. Ein System ist somit eine Zusammenstellung von einzelnen (neu entwickelten oder vorgefertigten) Komponenten, die miteinander interagieren.

Für die Qualität eines solchen Systems ist zunächst wichtig, dass die einzelnen Komponenten fehlerfrei funktionieren. Dies kann durch einen *Komponententest* überprüft werden. Die Zusammenstellung mehrerer Komponenten, die einzeln störungsfrei funktionieren, garantiert allerdings nicht, dass das daraus resultierende Gesamtsystem auch korrekt funktionieren wird. Eine sehr ernstzunehmende, aber meistens unterschätzte Fehlerquelle ist die Komponenteninteraktion [SJ04]. Zum Zweck der Entdeckung fehlerhafter Komponenteninteraktionen eignet sich der *Integrationstest*. Eine systematische Herangehensweise an die Qualitätssicherung ist folglich besonders auf Komponenten- und Integrationsebene von sehr hoher Bedeutung. Um die Wahrscheinlichkeit der Entdeckung von Komponenten- und Interaktionsfehlern zu erhöhen, müssen die im Rahmen dieser zwei Testphasen generierten Testfallmengen den Quelltext der Software möglichst vollständig abdecken.

Damit solche Fehler bei der Implementierung gar nicht erst entstehen, sind vorbeugende Maßnahmen zu treffen. Dazu zählt die systematische Überprüfung aller Artefakte, die im Laufe des Softwarelebenszyklus vor der Implementierungsphase entstehen (z. B. Anforderungsspezifikation oder Entwurfs-Modelle). Somit können Fehler frühzeitig entdeckt und behoben werden, was für den Projekterfolg von entscheidender Bedeutung ist, da die Kosten für die Behebung eines Fehlers umso geringer ausfallen, je früher der Fehler entdeckt wird. Um Entwurfs-Modelle zu hinterfragen gibt es mehrere Methoden; unter anderem können zu diesem Zweck *modellbasierte Testfälle* eingesetzt werden. Dies sind Testfälle, die anhand des Modells generiert werden und es möglichst vollständig überdecken. Neben der Überprüfung der Modelle kann mittels solcher Testfälle der Quelltext der Software dahingehend untersucht werden, ob das im Modell beschriebene Verhalten adäquat umgesetzt wird. Bei dieser Vorgehensweise muss aber beachtet werden, dass das Modell lediglich eine abstrakte Beschreibung des beabsichtigten Verhaltens darstellt. Deshalb wird im Allgemeinen bei der Ausführung modellbasierter Testfälle zur Überprüfung der Implementierung nur eine grobe Überdeckung des Programmcodes erreicht. Das bedeutet, dass solche Testfälle allein nicht ausreichend sind, um den Quelltext der Software gründlich zu überprüfen.

Für eine möglichst gründliche und kosteneffiziente Gestaltung des Testprozesses, sollte dieser weitgehend automatisiert durchgeführt werden. Da die Testfallgenerierung eine der wichtigsten und aufwändigsten Aktivitäten im Testprozess darstellt, ist vor allem die Automatisierung

dieses Schrittes von besonderem Interesse. Neben der Testfallgenerierung kann auch die Testfallausführung eine sehr zeitintensive Aktivität darstellen, wobei besonders die Einstufung der Ergebnisse jedes einzelnen Testfalls mit hohem manuellem Aufwand verbunden ist. Angesichts dieser Tatsache ist es wichtig, dass während der Testfallgenerierungsphase neben dem Erreichen einer möglichst hohen Überdeckung des Testobjekts auch ein möglichst geringer Umfang der Testfallmenge angestrebt wird.

1.2. Zielsetzungen

Unter Berücksichtigung der oben beschriebenen Aspekte, wurden im Rahmen der vorliegenden Arbeit folgende Zielsetzungen verfolgt:

Unterstützung der automatischen Komponententestgenerierung und Optimierung

Das werkzeuggestützte Verfahren, das im Rahmen dieser Arbeit beschrieben wird, ermöglicht die automatische modellbasierte Komponententestgenerierung zur Erfüllung etablierter modellbasierter Überdeckungskriterien für den Komponententest (*Zustands-*, *Transitions-* und *Transitionspaarüberdeckung*). Durch Benutzung genetischer Algorithmen erlaubt das Werkzeug neben der automatischen *Generierung* auch die *Optimierung* der Testfallmengen hinsichtlich zweier Ziele:

- einerseits hinsichtlich einer möglichst hohen Überdeckung bezüglich des angestrebten Kriteriums und
- anderseits hinsichtlich einer möglichst geringen Anzahl an Testfällen.

Die Ergebnisse, die durch Anwendung des Werkzeugs bei der Komponententestgenerierung erreicht wurden, werden in Abschnitt 6.3 beschrieben. Des Weiteren unterstützt das Werkzeug die *Visualisierung* der durch Komponententestfälle überdeckten und noch zu überdeckenden Entitäten (siehe Abschnitt 5.2) und die *modellbasierte Regressionstestgenerierung* für den Komponententest (siehe Abschnitt 5.3).

Beschreibung modellbasierter Überdeckungskriterien für den Integrationstest

Existierende Ansätze zur automatischen Generierung von Integrationstestfällen anhand von Modellbeschreibungen betrachten Sequenzen von Aufrufen zwischen mehreren Komponenten, ohne

die internen Zustände der aufrufenden und aufgerufenen Komponenten zu berücksichtigen. Deshalb werden zunächst im Rahmen dieser Arbeit (in Abschnitt 3.3.2) eine Reihe so genannter *zustandsbasierter Integrationstestüberdeckungskriterien* beschrieben, welche die internen Zustände der interagierenden Komponenten berücksichtigen. Da die unterschiedlichen Kriterien mit unterschiedlichem Testaufwand verbunden sind, wird zu jedem einzelnen Kriterium eine Formel angegeben, die es erlaubt, die Anzahl der zu überdeckenden Entitäten zu bestimmen. Schließlich werden alle Kriterien in eine Subsumptionshierarchie eingeordnet. Die betrachteten zustandsbasierten Kriterien wurden in [SOP07] eingeführt.

Unterstützung der automatischen Integrationstestgenerierung und Optimierung

Neben der Komponententestgenerierung unterstützt das vorhin erwähnte Werkzeug auch die *Generierung* und *Optimierung modellbasierter Integrationstestfälle*, welche zustandsbasierte Integrationstestüberdeckungskriterien erfüllen. Die Ergebnisse, die durch Anwendung genetischer Algorithmen bei der Integrationstestgenerierung erreicht wurden, werden in Abschnitt 6.4 beschrieben. Auch für Integrationstestfälle kann mittels des Werkzeugs visualisiert werden, welche Entitäten überdeckt wurden und welche Entitäten noch zu überdecken sind. Diese Funktionalität wird in Abschnitt 5.2 beschrieben.

Bewertung modellbasierter Testfälle hinsichtlich ihres Fehlererkennungspotentials bezüglich Modellierungs- und Implementierungsfehler

Eine weitere Zielsetzung dieser Arbeit ist die Bewertung des Fehlererkennungspotentials von Testfallmengen, die modellbasierte Überdeckungskriterien für den Komponententest oder Integrationstest erfüllen. Dazu wurde zunächst ein Mutationstest auf Modellebene durchgeführt, um zu bewerten, inwiefern *Modellfehler entdeckt werden* können (siehe Abschnitt 7.1). Um das Potential hinsichtlich der *Erkennung von Implementierungsfehlern* zu evaluieren, wurde eine modellbasiert generierte Integrationstestfallmenge auf eine Software für die Steuerung und Überprüfung von Patientenliegen ausgeführt (siehe Abschnitt 7.2).

All diese Zielsetzungen wurden im Rahmen des Forschungsprojekts *Unterstützung Inkrementeller Testdaten* (Kurz *UnITeD*) verfolgt. Dieses Projekt wurde im Auftrag des Bayerischen Wirtschaftsministeriums von den Verbundpartnern *Lehrstuhl für Software Engineering* und der *Afra GmbH* in Erlangen durchgeführt. Dabei entstand zwischen den Jahren 2006 und 2009 ein Verfah-

ren, das auf genetischen Algorithmen basiert und die vollautomatische Optimierung und Evaluierung modellbasierter Testfälle ermöglicht. Dieses Verfahren wurde in ein gleichnamiges Werkzeug umgesetzt.

Die entwickelte Testprozedur wurde erfolgreich im Bereich Magnetresonanztomographie beim Pilotpartner *Siemens Medical Solutions* in Erlangen erprobt. Hier wurden bis zum Zeitpunkt des Werkzeugeinsatzes Testfälle manuell anhand der Spezifikation erstellt und in Textdateien zusammengefasst. Besonders die manuelle Wartung der Testfälle war dabei sehr aufwändig: änderte sich eine Anforderung, so mussten auch mühsam alle Testfälle identifiziert werden, die in Folge der Änderung angepasst werden mussten.

Durch den Einsatz des Werkzeugs wurden der Testfallerstellungsprozess und die Wartung der generierten Testfallmengen bedeutend vereinfacht: manueller Aufwand entstand nur noch bei der Erstellung und der Anpassung der Modelle. Änderte sich etwas an dem zugrundeliegenden Modell, so wurden die davon betroffenen Testfälle automatisch durch das Werkzeug ermittelt und gegebenenfalls neue Testfälle generiert. Am Ende des Piloteinsatzes betrug der Anteil automatisch generierter Testfälle 70% der Gesamtanzahl erstellter Testfälle, was auf eine hohe Akzeptanz des Werkzeugs seitens des Pilotpartners hindeutet.

Ein wesentlicher Beitrag der hier vorgestellten Forschungsarbeit besteht also in der Entwicklung eines Verfahrens zur automatischen Testfallgenerierung für den Komponenten- und Integrationstest, an dem der Autor maßgeblich beteiligt war. Hierzu konnte auf am Lehrstuhl für Software Engineering bereits vorhandene Erfahrungen im Bereich der Optimierung mittels genetischer Algorithmen zurückgegriffen werden. Des Weiteren basiert diese Arbeit auch auf bereits etablierte modellbasierte Komponententestkriterien sowie auf am Lehrstuhl für Software Engineering entstandene modellbasierte Integrationstestkriterien.

Der Hauptbeitrag dieser Arbeit besteht in der Bewertung modellbasierter Testfallmengen für den Komponenten- und Integrationstest hinsichtlich ihres Fehlererkennungspotentials. Hier wurden neue Mutationsoperatoren auf Modellebene konzipiert und umgesetzt, um zu evaluieren, inwiefern automatisch generierte Testfallmengen die Entdeckung von Modellfehlern unterstützen. Darüber hinaus wurde modellbasiertes Testen mit konventionellem Testvorgehen im Hinblick auf ihr Potential zur Erkennung von Implementierungsfehlern verglichen.

1.3. Gliederung

Diese Ausarbeitung ist wie folgt gegliedert. Zunächst wird in Kapitel 2 Grundlagenwissen beschrieben, auf dem diese Arbeit aufbaut. Abschnitt 2.1 führt das modellbasierte Softwareentwicklungsparadigma ein, gefolgt von der Beschreibung der weit verbreiteten Modellierungssprache UML in Abschnitt 2.2. Im darauf folgenden Abschnitt 2.3 wird ein zur modellbasierten Entwicklung orthogonaler Ansatz beschrieben, und zwar die komponentenbasierte Softwareentwicklung. Danach werden in Abschnitt 2.4 Software-Qualitätssicherungsverfahren im Allgemeinen vorgestellt. Der Abschnitt 2.5 beschreibt eines der wichtigsten Qualitätssicherungsverfahren, und zwar das Testen. Des Weiteren wird in diesem Abschnitt die Basisterminologie dieser Arbeit (Testfall, Testsuite usw.) eingeführt und verschiedene Testphasen (in Unterabschnitt 2.5.1) und Testarten (in Unterabschnitt 2.5.2) vorgestellt. Das modellbasierte Testen wird im letzten Abschnitt (2.6) des Kapitels eingeführt.

Nach dem allgemeinen Einstieg wird in Kapitel 3 auf modellbasierte Überdeckungskriterien eingegangen. Zur Modellierung des Verhaltens von Komponenten wurden UML-Zustandsautomaten eingesetzt; diese werden in Abschnitt 3.1 beschrieben. Im darauf folgenden Abschnitt 3.2 werden Überdeckungskriterien für einzelne Zustandsautomaten vorgestellt. Auf modellbasierte Schnittstellenüberdeckungskriterien wird in Abschnitt 3.3 eingegangen, gefolgt von der Beschreibung so genannter zustandsbasierter Integrationstestkriterien in Unterabschnitt 3.3.2.

Kapitel 4 beschäftigt sich mit genetischen Algorithmen. Auf die Notwendigkeit des Einsatzes genetischer Algorithmen zur automatischen Generierung von Testfällen wird in Abschnitt 4.1 eingegangen, gefolgt von der Beschreibung der allgemeinen Funktionsweise solcher Algorithmen in Abschnitt 4.2. Danach wird in Abschnitt 4.3 die Anpassung der genetischen Algorithmen zum Zweck der automatischen, modellbasierten Testfallgenerierung erläutert.

Anschließend wird in Kapitel 5, Abschnitt 5.1 das Werkzeug UnITeD beschrieben, das genetische Algorithmen einsetzt und dadurch Testfälle automatisch zu generieren und zu optimieren erlaubt. Weitere nützliche Funktionen des Werkzeugs, und zwar die Visualisierung der erreichten Modellüberdeckung und die Regressionstestgenerierung werden in jeweils einem Abschnitt (5.2 bzw. 5.3) dieses Kapitels vorgestellt.

Kapitel 6 stellt die Ergebnisse der Testfallgenerierung dar. Zunächst werden dazu in Abschnitt 6.1 die evaluierten Anwendungen beschrieben, gefolgt von der Beschreibung der Parametrisierung der Algorithmen in Abschnitt 6.2. Die Ergebnisse der Komponententestgenerierung werden

in Abschnitt 6.3 vorgestellt. Daraufhin beschreibt Abschnitt 6.4 die Ergebnisse der Integrationstestgenerierung.

Die Evaluierung des Fehlererkennungspotentials modellbasierter Testfälle wird in Kapitel 7 beschrieben. Einerseits wurde mittels eines Modell-Mutationstests evaluiert, inwiefern modellbasierte Testfallmengen erlauben, Modellierungsfehler zu erkennen; die Ergebnisse werden in Abschnitt 7.1 beschrieben. Andererseits wurde evaluiert, ob solche Testfälle auch die Erkennung von Implementierungsfehlern unterstützen, was in Abschnitt 7.2 dargestellt wird.

Abschließend gibt das Kapitel 8 eine Zusammenfassung der vorliegenden Arbeit zusammen mit einem Ausblick.

2. Grundlagen

Dieses Grundlagenkapitel beschreibt zunächst wesentliche Aspekte der modellbasierten Softwareentwicklung (Abschnitt 2.1) zusammen mit der weit verbreiteten Modellierungssprache UML (Abschnitt 2.2). Mittels UML-Diagrammen können unter anderem komponentenbasierte Systeme beschrieben werden, die in Abschnitt 2.3 eingeführt werden. Darauf folgend werden Software-Qualitätssicherungsverfahren im Allgemeinen vorgestellt (Abschnitt 2.4), gefolgt von der detaillierten Betrachtung eines der wichtigsten Repräsentanten solcher Verfahren, dem Testen (Abschnitt 2.5). Abschließend wird eine spezielle Testart besonders fokussiert, und zwar der modellbasierte Test (Abschnitt 2.6).

2.1. Modellbasierte Softwareentwicklung

Der Begriff *modellbasierte Softwareentwicklung* beschreibt ein Softwareentwicklungsparadigma, bei dem Modelle als zentraler Bestandteil betrachtet werden und während des gesamten Softwarelebenszyklus durchgängig verwendet werden. Im Kontext dieser Arbeit ist ein Modell eine abstrakte Beschreibung eines Softwaresystems, welche relevante Aspekte der Struktur bzw. des Verhaltens des Systems darstellt.

Als Spezialisierung dieses Paradigmas betont die *modellgetriebene Softwareentwicklung* speziell den generativen Aspekt und wird in [TP07] wie folgt definiert: „Modellgetriebene Softwareentwicklung befasst sich mit der Automatisierung in der Softwareherstellung. Dies bedeutet, dass möglichst viele Artefakte eines Softwaresystems generativ aus formalen Modellen abgeleitet werden. Bei den erzeugten Artefakten handelt es sich nicht nur um den Quelltext einer ausführbaren Programmiersprache (z. B. Java oder C++), sondern auch um andere Dateien, die für die Lauffähigkeit des Systems benötigt werden, z. B. Konfigurationsdateien, Resource Bundles und Datenbankskripte. Darüber hinaus können auch den Entwicklungsprozess unterstützende

Artefakte generiert werden, z. B. Softwaretests und Dokumentationen.“

Dank der guten Werkzeugunterstützung bei der Erstellung von Modellen können im Rahmen der modellgetriebenen Softwareentwicklung aus Artefakten einer spezifischen Phase, Artefakte generiert werden, die in den darauf folgenden Phasen benutzt werden können. So z. B. bieten Modellierungswerkzeuge die Möglichkeit, aus Modellen automatisch Codegerüste[1] zu generieren. Um die erwünschte Funktionalität zu erhalten, müssen diese Codegerüste in einem weiteren Schritt verfeinert werden, indem das Verhalten (Implementierung der Methoden) manuell codiert wird.

Aufgrund der zentralen Bedeutung der Modelle hängt der Erfolg modellbasiert entwickelter Softwaresysteme sehr davon ab, ob die erstellten Modelle eine hohe Qualität erreichen, d. h. ob sie die zu modellierenden Aspekte korrekt und vollständig wiedergeben und dabei verständlich und überschaubar bleiben. [RBGW10] widmet sich grundlegenden Betrachtungen zur Qualität von Modellen; zu den darin erwähnten Qualitätseigenschaften zählen: Korrektheit, Einfachheit, Verständlichkeit, Angemessenheit, Änderbarkeit, Vollständigkeit, Widerspruchsfreiheit und Prüfbarkeit. Zur Überprüfung von Modellen hinsichtlich dieser Eigenschaften gibt es eine Reihe systematischer Qualitätssicherungsverfahren[2].

In erster Linie sollte ein Modell *korrekt* sein, da Fehler, die bei der Modellierung entstehen und unentdeckt bleiben, sich auch in den aus dem Modell generierten oder anhand des Modells manuell erstellten Artefakten wiederfinden lassen. Allgemein bekannt ist, dass die Behebung solcher Fehler in den späteren Phasen des Softwarelebenszyklus (z. B. Implementierungs- oder Testphasen) das Vielfache der Kosten verursacht, die bei einer Behebung während der Modellierungsphase entstanden wären [GVEB08].

Für die Erstellung von Modellen gibt es eine Vielzahl von Modellierungssprachen. Abhängig von dem zugrunde liegenden Vokabular unterscheidet man zwischen *visuellen* und *textuellen Sprachen*, wobei es auch Sprachen gibt, die sowohl textuelle als auch visuelle Konstrukte beinhalten. Textuelle Sprachen haben als Vokabular Wörter der natürlichen Sprachen. Dagegen stellt das Vokabular der visuellen Modellierungssprachen geometrische Formen (z. B. Rechtecke, Kanten, Pfeile) zur Verfügung. Durch Zusammensetzung solcher Formen werden Diagramme erstellt. Jedes Diagramm hebt bestimmte Aspekte des Systems hervor; die Menge aller Diagramme stellt ein Modell des Systems dar.

[1] Z. B. Java-Klassen mit Attributen und Methodensignaturen

[2] Siehe Abschnitt 2.4

Die Semantik einer Sprache ordnet jeder graphischen oder textuellen Entität eine konkrete Bedeutung zu. Je nach Grad ihrer semantischen Präzision lassen sich Modellierungssprachen wie folgt klassifizieren:

- *informale Modellierungssprachen*: zu den bekanntesten Vertretern der informalen Sprachen zählen natürliche Sprachen, wie z. B. Deutsch oder Englisch. Der Vorteil der Benutzung solcher Sprachen besteht darin, dass diese Sprachen keinen zusätzlichen Lernaufwand verursachen und dabei sehr ausdrucksmächtig sind. Zu den Nachteilen zählen ihre Mehrdeutigkeit und die Tatsache, dass sie schlecht analysierbar sind. Deshalb eignen sich solche Sprachen meistens nicht als Mittel zur Beschreibung komplexer Sachverhalte im Kontext der Softwareentwicklung.
- *semi-formale Modellierungssprachen*: Beispiele solcher Sprachen sind *Entity-Relationship-Diagramme*[3] oder die *Unified Modeling Language*[4]. Diese stellen einen Kompromiss zwischen informalen und formalen Sprachen dar und enthalten somit sowohl Teile der natürlichen Sprachen als auch Elemente mit eindeutiger Semantik.

  Im Gegensatz zu informalen Sprachen ermöglichen sie somit die Erstellung einer Spezifikation, die kaum unterschiedliche Interpretationen zulässt. Des Weiteren ist die meist visuelle Form dieser Sprachen vorteilhaft, da Menschen komplexe Strukturen leichter verstehen können, wenn diese mittels Diagrammen (an Stelle von rein textuellen Notationen) beschrieben werden. Bei Einsatz semi-formaler Sprachen ist zu beachten, dass dies eine ausreichende Schulung von Mitarbeitern voraussetzt, was mit Kosten verbunden ist.
- *formale Modellierungssprachen*: z. B. *endliche Zustandsautomaten*[5], *Petri-Netze*[6]. Diese Sprachen haben eine eindeutige Syntax und Semantik, was dazu führt, dass Modelle, die mit Hilfe solcher Sprachen beschrieben wurden, eindeutig sind und keinen Spielraum für Interpretationen erlauben.

  Des Weiteren ist die Erfüllung wichtiger Eigenschaften (z. B. Verklemmung) oft beweisbar und auch durch den Einsatz von Werkzeugen automatisch durchführbar. Allerdings entsteht im Vergleich zu semi-formalen Sprachen ein noch höherer Einarbeitungsaufwand

[3]Diese Sprache eignet sich besonders zur Datenmodellierung im Kontext von Datenbankanwendungen
[4]Kurz *UML*; diese Sprache wird in Abschnitt 2.2 genauer beschrieben
[5]Engl. *Finite State Machine*, Kurz *FSM*
[6]Diese Sprache eignet sich besonders für die Modellierung nebenläufiger Prozesse mit Parallelisierung und Synchronisation

für die Mitarbeiter und dadurch noch höhere Kosten. Auch die Erstellung solcher Modelle ist, im Vergleich zu semi-formalen Modellierungssprachen, aufwändiger.

Unabhängig von der Syntax oder Semantik wird noch unterschieden, ob es sich um Sprachen handelt, die nur in einer bestimmten Domäne einsetzbar sind, also so genannte *Domain Specific Languages* (Kurz *DSL*) oder um Sprachen, die zur Beschreibung allgemeiner Problemstellungen geeignet sind: *General Purpose Languages* (Kurz *GPL*).

2.2. Unified Modeling Language

Die *Unified Modeling Language* (Kurz *UML*) ist eine Sprache, die sich besonders gut für die Modellierung objektorientierter Softwaresysteme eignet. Diese Sprache erfreut sich in letzter Zeit immer größerer Akzeptanz, was mehrere Gründe hat. Zum einen ist sie als General Purpose Language nicht auf eine bestimmte Softwaredomäne eingeschränkt und zum anderen kann sie durch so genannte *Profile* auf bestimmte Anwendungsgebiete zugeschnitten werden. Ein Beispiel für so ein Profil ist das *UML 2.0 Testing Profile* (Kurz *U2TP*) [BDG$^+$04]. Dieses Profil erlaubt die Spezifikation von Artefakten, die für die Testphase relevant sind (so z. B. die Modellierung des *System Under Test* (Kurz *SUT*)).

Die UML unterstützt die modellbasierte Softwareentwicklung, indem sie es ermöglicht, sowohl die statische Struktur als auch das dynamische Verhalten der Software zu modellieren. Weiterhin kann diese Sprache als *semi-formale Sprache* eingestuft werden, da sie zum einen Sprachkonstrukte mit eindeutiger Semantik zur Verfügung stellt, wie z. B. Aktivitätsdiagramme, welche auf der Semantik der Petri-Netze aufbauen. Auf der anderen Seite erlaubt sie auch die Benutzung natürlicher Sprache.

Die Verfügbarkeit vieler UML-Modellierungswerkzeuge stellt einen weiteren Grund für die große Verbreitung dieser Sprache dar. Diverse Werkzeuge wie z. B. *MagicDraw*[7] oder *Enterprise Architect*[8] bieten graphische Editoren, die die Modellierung sehr gut unterstützen. Zusätzlich stellen diese Werkzeuge Funktionalitäten zur Verfügung, die für die modellgetriebene Entwicklung sehr wichtig sind, wie z. B. automatische Codegenerierung.

Die geschichtliche Entwicklung der UML wird in [BRJ06] beschrieben. Durch die Anfang

[7]http://www.magicdraw.com/
[8]http://www.sparxsystems.de/

der 90er Jahre immer stärker aufkommende objektorientierte Softwareentwicklung entstand auch ein Bedarf an geeigneten Modellierungsmethoden und Sprachen. Somit entstanden viele Methodenvorschläge, unter anderem *Object Oriented Analysis and Design* von Grady Booch, *Object Modeling Technique* von Jim Rumbaugh und *Object Oriented Software Engineering* von Ivar Jacobsen. Diese Vorschläge setzten sich nicht durch, was letztendlich dazu führte, dass sich die genannten Autoren Mitte der 90er Jahre der Definition einer Standardmodellierungssprache widmeten. Dabei entstand die UML-Version 0.9, die 1996 veröffentlicht wurde. Die darauf folgende Version 1.0 wurde bei der *Object Management Group* (Kurz *OMG*) zur Standardisierung eingereicht. Daraufhin wurde eine überarbeitete Version der UML (Version 1.1) von der OMG als Standardnotation für die Analyse und Design von objektorientierten Programmen akzeptiert. Als relativ junge Modellierungssprache wurde die UML immer wieder geändert; die aktuelle Version der UML ist die 2.3[9], die im Mai 2010 veröffentlicht wurde [OMG10].

Verwaltet wird die UML-Spezifikation von der OMG, welches ein 1989 gegründetes Konsortium aus über 800 Unternehmen ist, das sich mit der Entwicklung von Standards beschäftigt. Neben vielen Softwareherstellern wie Microsoft, IBM oder Sun befinden sich darunter auch Unternehmen, deren Kerngeschäft nicht im Softwarebereich liegt, wie z. B. Daimler oder die Nasa. Ab der Version 2.0 wurde die UML-Spezifikation unterteilt in die:

- *UML Infrastructure Specification*: stellt das Fundament der Sprache dar und beschreibt die grundlegenden Sprachkonstrukte der UML, wie z. B. Klassen oder Assoziationen. Mit Hilfe dieser Konstrukte wird die UML Superstructure erfasst.
- *UML Superstructure Specification*: diese baut auf die Infrastructure auf und beschreibt Spracheinheiten, die sehr UML spezifisch sind, wie z. B. Anwendungsfall oder Aktivität.

Die UML enthält 14 Diagrammarten, welche in [RHQ$^+$07] unter Berücksichtigung der UML-Spezifikation übersichtlich und verständlich beschrieben werden. Diese Diagramme lassen sich in *Struktur-* (Engl. *Structure Diagrams*) und *Verhaltensdiagramme* (Engl. *Behavior Diagrams*) aufteilen. Alle verfügbaren Diagramme werden in Abbildung 2.1 dargestellt.

Die Strukturdiagramme eignen sich zur Modellierung der statischen Struktur eines Systems. Die Menge der Strukturdiagramme umfasst folgende Elemente:

- *Klassendiagramm* (Engl. *Class Diagram*): ist eines der meist benutzten Strukturdiagramme der UML. Es erlaubt die Modellierung von Klassen und deren Beziehungen unterein-

[9]Die aktuelle Version der UML-Spezifikation ist verfügbar unter: http://www.omg.org/spec/UML/Current

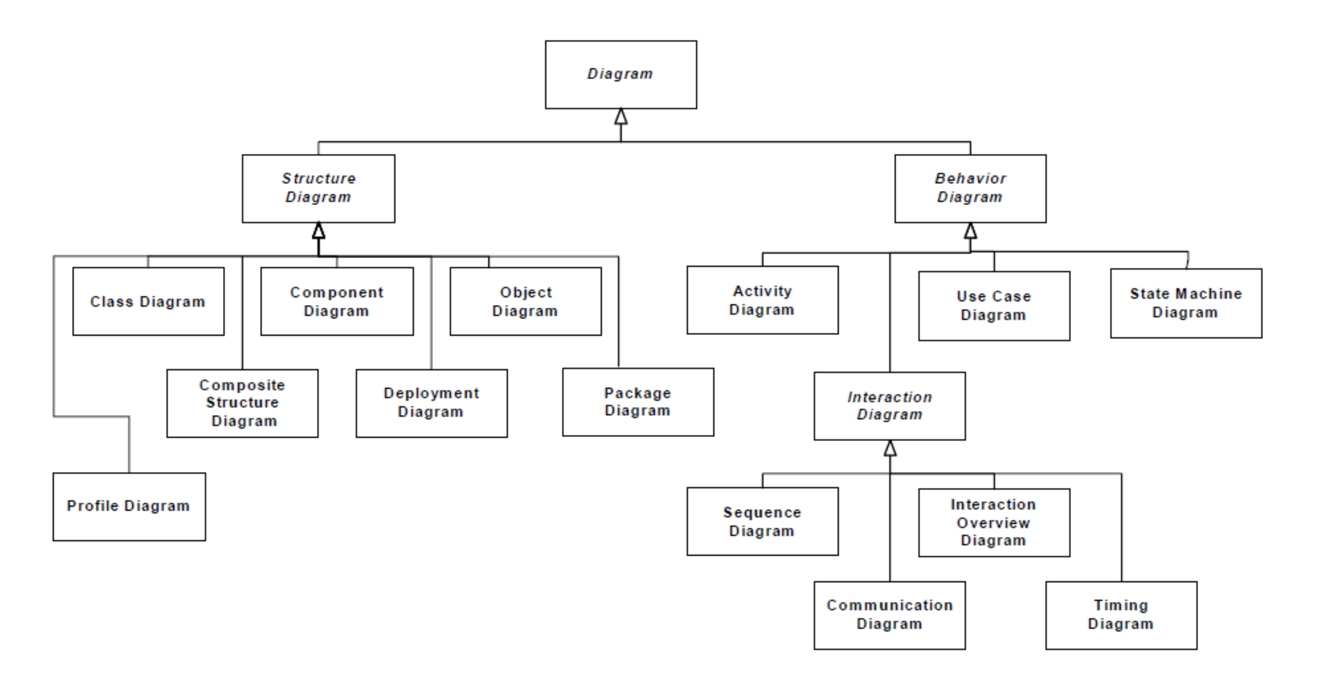


Abbildung 2.1.: Die Diagramme der UML (aus [OMG10], s. 720)

ander.

- *Profildiagramm* (Engl. *Profile Diagram*): eine der Stärken der UML ist, dass sie sehr gut mittels Profilen an die spezifischen Anforderungen des Kunden angepasst werden kann. Das Profildiagramm erlaubt die Beschreibung und Gruppierung eigendefinierter Stereotypen.
- *Paketdiagramm* (Engl. *Package Diagram*): da Modelle mit vielen Klassen schnell unübersichtlich werden können, erlaubt dieses Diagramm, eine übersichtliche Darstellung des Systems, indem Pakete[10] und deren Beziehungen dargestellt werden.
- *Objektdiagramm* (Engl. *Object Diagram*): zur Laufzeit von objektorientierten Anwendungen werden aus Klassen Objekte erzeugt. Dieses Diagramm erlaubt die Darstellung der während der Laufzeit existierenden Objekte mit ihren Links[11] und Attributwerten. Somit wird ein so genannter Schnappschuss eines Systems zu einem bestimmten Zeitpunkt dargestellt.
- *Kompositionsstrukturdiagramm* (Engl. *Composite Structure Diagram*): erlaubt die Darstel-

[10] Ist eine Ansammlung von Modellelementen (z. B. Klassen) wobei innerhalb dieser Strukturierungseinheit der Name der Elemente eindeutig sein muss

[11] Ist eine Instanz einer Assoziation

lung der internen Struktur eines Classifiers, sowie der Beziehungen mit anderen interagierenden Systembestandteilen. Dabei ist ein Classifier eine abstrakte Metaklasse, die eine ganze Gruppe von Modellelementen repräsentiert, die gleiche Eigenschaften haben. Zu diesen Elementen gehören z. B.: Klasse, Aktivität, Assoziation.

- *Komponentendiagramm* (Engl. *Component Diagram*): ermöglicht es die Struktur des Systems als Menge von Komponenten darzustellen. Der Fokus liegt dabei nicht auf der Beschreibung der inneren Funktionsweise oder dem strukturellem Aufbau der Komponente, sondern auf den Beziehungen zwischen mehreren Komponenten des Systems.
- *Verteilungsdiagramm* (Engl. *Deployment Diagram*): stellt die Zuordnung von Softwareartefakten auf die Ziel-Hardware dar. Somit wird die Softwaresicht um die benötigten Hardware-Komponenten vervollständigt.

Neben der Beschreibung der Struktur des zu realisierenden Systems mittels Strukturdiagrammen, kann mittels UML Verhaltensdiagrammen die Dynamik des Systems erfasst werden. Zu dieser Klasse zählen folgende Diagramme:

- *Use-Case-Diagramm* oder *Anwendungsfalldiagramm* (Engl. *Use Case Diagram*): stellt die Funktionalität des Systems grob durch eine Menge von Anwendungsfällen dar. Dabei beschreibt ein Anwendungsfall eine mögliche Benutzung des Systems. Des Weiteren werden die so genannten Akteure dargestellt, die mit dem System interagieren und Funktionen des Systems aufrufen. Beispiele für Akteure sind Personen oder andere Systeme.
- *Aktivitätsdiagramm* (Engl. *Activity Diagram*): beschreibt die Abläufe in einem System. Dazu werden mit Knoten einzelne Aktionen des Systems assoziiert. Diese Knoten werden durch gerichtete Kanten verbunden, die die Reihenfolge definieren, in der die einzelnen Aktionen durchgeführt werden.
- *Zustandsautomat*[12] (Engl. *State Machine Diagram*): beschreibt die einzelnen Zustände eines Systems und die Übergänge zwischen ihnen. Dieses Diagramm ist für diese Arbeit sehr wichtig und wird detailliert in Abschnitt 3.1 beschrieben.
- *Sequenzdiagramm* (Engl. *Sequence Diagram*): gehört zur Klasse der *Interaktionsdiagramme*[13]. Sequenzdiagramme eignen sich besonders für die Darstellung der Kooperation in

[12] Im Rahmen dieser Arbeit werden auch die Begriffe *Zustandsdiagramm* oder *Zustandsmaschine* benutzt

[13] Diese sind auch Verhaltensdiagramme und eignen sich besonders gut für die Modellierung von Interaktionen zwischen mehreren Objekten

Form von gesendeten und empfangenen Nachrichten mit besonderer Betonung der zeitlichen Abfolgen. Eine weitere Stärke ist die Darstellungsmöglichkeit alternativer Abläufe.

- *Kommunikationsdiagramm*[14] (Engl. *Communication Diagram*): gehört auch zu den Interaktionsdiagrammen und erlaubt eine ähnliche Darstellung wie die Sequenzdiagramme allerdings mit Fokus auf die statischen Beziehungen zwischen den interagierenden Objekten. Die zeitliche Abfolge der Nachrichten wird durch Nummerierung angegeben.
- *Timingdiagramm* oder *Zeitverlaufsdiagramm* (Engl. *Timing Diagram*): dieses Interaktionsdiagramm erlaubt die Darstellung von Zeitverlaufskurven von Zuständen. Mit Hilfe dieses Diagramms werden die Zustände beschrieben, in denen sich ein oder mehrere Classifier zu einem bestimmten Zeitpunkt befinden können.
- *Interaktionsübersichtsdiagramm* (Engl. *Interaction Overview Diagram*): zur Darstellung der Abläufe mehrerer Interaktionsdiagramme eignet sich dieses Diagramm. Vom Aufbau ähnlich wie das Aktivitätsdiagramm, enthält es in den Knoten allerdings keine einzelnen atomaren Aktivitäten, sondern ganze Interaktionsdiagramme.

Neben der UML hat die OMG viele weitere Standards entwickelt[15], die für die Softwareindustrie von großer Bedeutung sind, z. B.:

- *Object Constraint Language* (Kurz *OCL*): diese Sprache erlaubt es *Einschränkungen* (Engl. *Constraints*) für bestimmte Elemente der UML festzulegen. Beispielsweise erlaubt die Sprache, die Angabe von Invarianten für Attribute von Klassen oder die Formulierung von Vor- und Nachbedingungen für Methoden.
- Die Standards *XML*[16] *Metadata Interchange* (Kurz *XMI*) und *UML Diagram Interchange* (Kurz *UMLDI*) unterstützen den Austausch von Modellen zwischen verschiedenen UML-Werkzeugen.
- *Model Driven Architecture* (Kurz *MDA*): stellt den Standard der OMG zu modellgetriebenen Entwicklung dar. Im Rahmen dieses Standards wird eine konkrete Vorgehensweise für die modellgetriebene Entwicklung vorgeschlagen, die besonders auf die Trennung zwischen fachlichen und technischen Modellen Wert legt.

[14] Hieß in älteren UML-Versionen *Kollaborationsdiagramm*

[15] Diese sind alle über die OMG-Seite: http://www.omg.org verfügbar

[16] Kurz für *Extensible Markup Language*

2.3. Komponentenbasierte Softwareentwicklung

Im Rahmen eines komponentenbasierten Entwicklungsprozesses wird die zu erstellende Software nicht als ein monolithisches System von Grund auf neu entwickelt, sondern als Menge von interagierenden Komponenten zusammengestellt. Die einzelnen Komponenten können dabei entweder neu entwickelt oder von Drittanbietern zugekauft werden, wobei es sich im zweiten Fall um so genannte *Off-the-shelf*-Komponenten handelt.

Definition 2.1 (Komponente, laut IEEE Standard 610-1991 [IEE91]) *One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components. Note: The terms "module," "component," and "unit" are often used interchangeably or defined to be subelements of one another in different ways depending upon the context.*

Im weiteren Verlauf dieser Arbeit werden die Begriffe *Komponente*, *Software-Komponente*, *Modul* oder *Unit* als synonyme Bezeichnung für ein Softwarebaustein innerhalb eines Softwaresystems benutzt. Komponenten kommunizieren miteinander über ihre *Schnittstellen* (Engl. *Interfaces*), welche in [WCO03] wie folgt definiert werden: „the access points of components, through which a client component can request services declared in the interface and provided by another component.“

Neben der Tatsache, dass das komponentenbasierte Paradigma die Bewältigung komplexer Softwareprojekte überhaupt ermöglicht, bringt es weitere bedeutende Vorteile mit sich. Durch Benutzung von Komponenten (Off-the-shelf oder firmenintern entwickelt), die sich bereits im Betrieb bewährt haben, wird die Verlässlichkeit des Systems erhöht. Ein weiterer Vorteil ist, dass die Architektur einer aus mehreren kommunizierenden Komponenten zusammengesetzten Software meist klar und übersichtlich ist.

Damit die Vorteile dieses Paradigmas voll zum Tragen kommen, müssen die einzelnen Komponenten und deren Abhängigkeiten bestimmte Anforderungen erfüllen. Diese Anforderungen beziehen sich auf den Grad der funktionalen Bindung innerhalb einer Komponente (die *Kohäsion* innerhalb der Komponente) und auf den Grad der Interaktion zwischen einzelnen Komponenten (die *Kopplung* zwischen Komponenten). Ersteres sollte für die einzelnen Komponenten so hoch wie möglich sein. Dagegen sollte der Grad der Interaktion zwischen mehreren Komponenten so gering wie möglich sein. Dadurch wird die Übersichtlichkeit, Wartbarkeit und Wiederverwendbarkeit des Systems gewahrt. Änderungen innerhalb der Komponente wirken sich nicht nach

außen (also auf die anderen Komponenten) aus, was die Gefahr minimiert, dass eine einzelne Änderung im Code viele Nebeneffekte in nicht geänderten Codestellen verursacht. Komponenten, die lose mit anderen Komponenten gekoppelt sind, können darüber hinaus leicht in anderen Anwendungen wiederverwendet werden, da sie nicht viele Abhängigkeiten zu anderen Komponenten besitzen.

Ein komponentenbasiert entwickeltes System mit hoch kohäsiven und gering gekoppelten Komponenten bietet also viele Vorteile; allerdings können Fehler, die aus Inkompatibilitäten der Komponenten resultieren dadurch nicht ausgeschlossen werden. Mit Inkompatibilitäten, die bei der Interaktion zwischen Komponenten auftreten können, beschäftigen sich die Arbeiten von [JS05] und [Jun07], wobei folgende Klassifikation vorgeschlagen wurde:

- *Syntaktische Inkonsistenzen*: diese Klasse beschreibt Probleme, die bei der Übersetzung des Quellcodes in Maschinencode entdeckt werden können. Ein Beispiel für solch ein Problem ist, wenn die Parameter an den Operationen der Schnittstellen unterschiedliche Datentypen haben.
- *Semantische Inkonsistenzen*: solche Inkonsistenzen treten auf, falls Daten, die zwischen Komponenten ausgetauscht werden, von den einzelnen Komponenten unterschiedlich interpretiert werden. So z. B. wenn eine Komponente einen bestimmten Integer-Wert als Kenngröße für das Gewicht in Maßeinheiten des metrischen Systems, wohingegen die andere Komponente diesen Wert in Maßeinheiten des amerikanischen Systems interpretiert.
- *Anwendungsbasierte Inkonsistenzen*: wenn Komponenten wiederverwendet werden, dann werden sie häufig in einer anderen, als der ursprünglich gedachten Anwendungslandschaft eingesetzt. Falls die Komponenten die Vorgaben der neuen Umgebung nicht erfüllen, dann bestehen anwendungsbasierte Inkonsistenzen. Beispiel für so eine Inkonsistenz ist, wenn eine Komponente in der neuen Umgebung mit Parameterwerten aufgerufen wird, die außerhalb des Wertebereichs liegen, der von der Komponente unterstützt wird.
- *Pragmatische Inkonsistenzen*: solche Inkonsistenzen betreffen Unstimmigkeiten in der Rechnerumgebung[17], in der die Komponente eingesetzt wird. Pragmatisch werden diese Inkonsistenzen deshalb genannt, weil es um die tatsächliche Ausführung unter Einfluss der Rechnerumgebung geht. Inkonsistenzen dieser Art treten z. B. auf, wenn die Komponente

[17] Dieser Begriff wird in der Arbeit [Jun07] stellvertretend für die komplette Hard- und Softwareumgebung der Komponente benutzt

die an sie gestellten Zeitanforderungen für die Erbringung eines Dienstes nicht erfüllen kann.

Zur Erkennung solcher Inkonsistenzen eignen sich Qualitätssicherungsverfahren, speziell der in Abschnitt 2.5.1 beschriebene *Integrationstest*. Falls solche Inkonsistenzen auftreten, müssen die betroffenen Komponenten entweder angepasst werden oder Mechanismen geschaffen werden, um diese Inkonsistenzen zu umgehen. Da die Anpassung besonders im Falle von Off-the-shelf-Komponenten nicht durchgeführt werden kann, müssen *Wrapper*[18] um die einzelnen Komponenten geschaltet werden [Jun07]. Ein Beispiel für eine zusätzliche Funktionalität, die durch Wrapper realisiert werden kann, ist die Transformation von Werten.

Im bisherigen Verlauf dieses Kapitels wurden zwei Entwicklungsparadigmen vorgestellt, die sich kombinieren lassen: so können mit Hilfe der Modellierungssprache UML komponentenbasierte Softwaresysteme entworfen werden. Mit den Komponentendiagrammen[19] gibt spezielle Diagrammtypen, die sich für die Beschreibung mehrerer Komponenten und deren Beziehungen eignen. Das Verhalten einzelner Komponenten kann mittels Verhaltensdiagrammen der UML beschrieben werden[20].

2.4. Software-Qualitätssicherungsverfahren

Die in diesem Abschnitt betrachteten Verfahren zielen auf die Überprüfung der Qualitätsmerkmale eines Softwaresystems. Dabei kann Softwarequalität aus zwei Blickwinkeln betrachtet werden. Die externe Qualität bezieht sich auf die Erfüllung von Eigenschaften, die von einem beliebigen Benutzer der Software bewertbar sind, wie z. B. *Korrektheit*, *Benutzerfreundlichkeit*, *Robustheit*, *Zuverlässigkeit*, *Verfügbarkeit*. Als interne Qualität bezeichnet man die Erfüllung von Eigenschaften, die nur von Personen bewertet werden können, die Einblick in den Quellcode der Komponente haben. Bespiele für solche Eigenschaften sind *Wartbarkeit*, *Wiederverwendbarkeit* oder *Portierbarkeit*.

Qualität hat also mehrere Dimensionen, die nicht alle gleich wichtig sind. Für unterschiedliche Domänen sind unterschiedliche Qualitätseigenschaften von besonderer Bedeutung; so z. B.

[18]Engl. für Hülle, bezeichnet eine Software die eine andere Software umhüllt, um bestimmte zusätzliche Funktionalitäten hinzuzufügen
[19]Siehe Abschnitt 2.2
[20]Siehe Abschnitt 3.1

ist für sicherheitskritische Steuerungssysteme eine hohe Zuverlässigkeit besonders wichtig, für Anwendersoftware dürfte dagegen meistens eine hohe Verfügbarkeit Vorrang haben. Unabhängig von der Domäne ist allerdings wichtig, dass die Software korrekt ist. Dabei unterscheidet man bei der Softwareentwicklung zwischen verschiedenen Arten von Software-Inkorrektheiten [Lyu96]:

- Ein *Mistake* (*Irrtum*) ist ein mentaler Vorgang, der dazu führt, dass Fehler entstehen.
- Ein *Fault* (*Softwarefehler*, *Fehler*) stellt eine Abweichung zwischen einem beabsichtigten und einem tatsächlich implementierten Softwareprodukt dar. Jeder Irrtum führt zwangsläufig zu dem Entstehen von Fehlern.
- Ein *Error* (*fehlerhafter Zustand*) stellt eine Abweichung zwischen dem beabsichtigten und dem realisierten internen Zustand dar, z. B. eine fehlerhafte Variablenbelegung. Ein Fehler kann zu einem fehlerhaften Zustand führen, muss aber nicht.
- Ein *Failure* (*Programmversagen*) stellt eine Abweichung zwischen der beabsichtigten und der realisierten Ausgabe des Programms dar. Ein fehlerhafter Zustand kann zu einem Programmversagen führen, muss aber nicht.

Das Schwierige beim Auffinden von *Fehlern* ist, dass diese nicht immer zur Folge haben, dass bei der Benutzung der Software ein fehlerhaftes – vom Benutzer oder Tester der Software beobachtbares – Verhalten auftritt. Dafür muss das fehlerhafte Programm bei der Ausführung erst einmal in einen *fehlerhaften Zustand* übergehen, was dann zu einem beobachtbaren *Programmversagen* führen kann.

Um zu vermeiden, dass solche Fehler überhaupt entstehen, kann eine ingenieurmäßige Herangehensweise an die Softwareentwicklung (Benutzung der Konstruktionsprinzipien Abstraktion, Teile und Herrsche, Orientierung an Vorgehensmodellen, Einsatz von Werkzeugen usw.) helfen. Eine solche ingenieurmäßige Herangehensweise bezeichnet man als *konstruktive Qualitätssicherung*.

Jedoch lassen sich trotz ingenieurmäßiger Softwareentwicklung aufgrund menschlichen Unvermögens Softwarefehler nicht ganz vermeiden. An dieser Stelle müssen *analytische Qualitätssicherungsverfahren* eingesetzt werden. Diese lassen sich in zwei Klassen aufteilen, je nachdem welches Ziel verfolgt wird:

- *Verifikation*: unter Verifikation versteht man die:

> „Überprüfung, ob die Ergebnisse einer Entwicklungsphase die Anforderungen zu Beginn der Phase erfüllen.“[Lig09]

Beispiele für Verifikationstätigkeiten sind der Komponenten-[21] und Integrationstest[22].

- *Validierung*: Validierung wird in [Lig09] wie folgt definiert:

  > „Überprüfung, ob ein Software-Produkt die Anforderungen und Bedürfnisse des Benutzers befriedigt.“

  Im Gegensatz zur Verifikation wird also im Rahmen der Validierungsaktivitäten überprüft, ob die vom Kunden tatsächlich gewünschte Funktionalität umgesetzt wurde. Beispiel für eine Validierungsaktivität ist, die Spezifikation dahingehend zu überprüfen, ob sie die Kundenwünsche korrekt und vollständig wiedergibt.

Unabhängig von dem konkreten Ziel lassen sich analytische Qualitätssicherungsverfahren[23] nach [Lig09] wie folgt kategorisieren:

- *statische Verfahren*: dies sind Verfahren, die eine Bewertung der Artefakte (z. B. Modelle oder Quelltext) vornehmen, ohne diese dabei auszuführen. Solche Verfahren sind somit zu jedem Entwicklungszeitpunkt anwendbar, werden meistens bewusst nicht automatisiert durchgeführt und binden den Menschen sehr stark mit ein.

  Beispiele solcher Verfahren sind *Inspections*, *Reviews* und *Walkthroughs*. Hierbei werden zur Fehlerfindung Artefakte durch Menschen begutachtet. Der Vorteil der Anwendung solcher Methoden besteht darin, dass sie nicht nur solche Fehler zu entdecken erlauben, die zu einem Programmversagen führen.

- *dynamische Verfahren*: im Gegensatz zu den statischen Verfahren benötigen die dynamischen Verfahren ausführbare Artefakte, wie z. B. den Quelltext der Software. Hier geht es nicht mehr um die Durchsicht bestimmter Artefakte, sondern um ihre Ausführung, um dann anhand der erzeugten Ausgabe zu entscheiden, ob sie die Anforderungen erfüllen. Mit Hilfe dieser Verfahren können nur Fehler entdeckt werden, die auch zu einem vom Menschen beobachtbaren Programmversagen führen.

Im Folgenden wird sich dieses Kapitel dem wichtigsten Repräsentanten der Klasse der dynamischen Verfahren widmen, und zwar dem Testen.

[21] Siehe Abschnitt 2.5.1.1
[22] Siehe Abschnitt 2.5.1.2
[23] In [Lig09] werden sie Prüftechniken genannt

2.5. Testen

Im IEE Standard 610.12-1990 [IEE90] wird der Begriff Testen wie folgt definiert:

Definition 2.2 (Test) *An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.*

Unter Testen versteht man also die Ausführung eines Programms in einer bestimmten Umgebung mit dem Ziel, das Programm bezüglich bestimmter Eigenschaften zu evaluieren. Beispiele für solche Eigenschaften wurden am Anfang des Abschnitts 2.4 beschrieben, wobei die Korrektheit als eine der wichtigsten Eigenschaften anzusehen ist. Deshalb ist das Testen in den meisten Fällen darauf ausgerichtet, Fehler im Programm zu finden. Dabei kann mittels Testen nachgewiesen werden, dass ein Programm für bestimmte Eingaben unter bestimmten Randbedingungen nicht funktioniert, also einen Fehler enthält. Da der Eingaberaum schon für kleine Programme sehr groß sein kann, kann das Programm meistens aus Zeit und Kostengründen nicht mit allen möglichen Eingaben unter allen möglichen Randbedingungen getestet werden. Aufgrund seines stichprobenartigen Charakters kann durch Testen nicht die Korrektheit von Programmen nachgewiesen werden. Vielmehr führt das Testen dazu, dass das Vertrauen in die Korrektheit des Programms wächst, falls bei der Ausführung der Testfälle keine Fehler entdeckt werden. Sehr bekannt ist in dieser Hinsicht die Aussage von Edsger Dijkstra in [Dij70]:

> „Program testing can be used to show the presence of bugs, but never to show their absence!“.

Ein Testverfahren, das nicht auf das Finden von Fehlern, sondern auf die Ableitung von Zuverlässigkeitsaussagen ausgerichtet ist, ist das so genannte *statistische Testen*. Hier geht es darum, durch Ausführung einer bestimmten Anzahl an korrekt ausgeführten Testfällen quantitative Aussagen über die Zuverlässigkeit des Systems zu treffen. Mit der quantitativen Bewertung der Softwarezuverlässigkeit komponentenbasierter Systeme mittels statistischer Testverfahren beschäftigt sich die Arbeit [Söh10].

In der vorliegenden Arbeit wird unter Testen das Verfahren gemeint, das auf die Aufdeckung von Fehlern in *reaktiven Softwaresystemen* ausgerichtet ist. Dies sind Systeme, die ständig auf die Aufrufe aus ihrer Umgebung reagieren, indem sie Ausgaben produzieren [Sch05]. Um solche Systeme zu testen, werden als Eingaben Testfälle benötigt. Ein *Testfall* ist eine Folge von

Aufrufen (im Verlauf der Arbeit auch *Teststeps* genannt), wobei jeder Aufruf aus einer *Nachricht* und – falls benötigt – aus *Parameterwerten* besteht[24]:

$$Testfall \;=\; Aufruf_1, ..., Aufruf_n,$$

$$wobei\ \forall i = 1..n:\ Aufruf_i = Nachricht_i([Parameterwerte_i])$$

Ein Testfall kann auch als *Testsequenz* mit dazugehörigen *Testdaten* beschrieben werden. Die Testsequenz ist die Folge der im Testfall enthaltenen Nachrichten. Da einzelne Nachrichten auch Parameter enthalten können, müssen für diese Parameter Werte festgelegt werden. Deshalb enthalten die Testdaten zu jeder Nachricht Werte für die Parameter.

Um nach der Ausführung der Testfälle zu bewerten, ob das dabei beobachtete Softwareverhalten angemessen ist, werden Informationen über das erwünschte Verhalten benötigt: diese Informationen werden als *Testorakel* bezeichnet. In der vorliegenden Definition von Testfall ist das Orakel nicht im Testfall mit enthalten. Für die Einstufung der Ergebnisse des Testlaufs[25], also für die Zuordnung eines *Verdicts* zu jedem einzelnen Testfall ist allein der ausführende Tester verantwortlich.

Ein Testfall ist im Allgemeinen dann von hoher Qualität, wenn er ein hohes Fehlererkennungspotential hat. Da ein Testfall, der eine hohe Überdeckung des Testobjekts erreicht, auch höhere Chancen bietet, Fehler zu finden (als ein Testfall, der eine niedrigere Überdeckung erreicht), gilt im Kontext dieser Arbeit ein Testfall dann als qualitativ hochwertig, wenn er das Modell, aus dem er generiert wurde, zu einem hohen Anteil überdeckt.

Eine Menge von Testfällen wird als *Testsuite* oder *Testfallmenge* bezeichnet. Laut [Ist10] ist eine Testsuite „a set of several test cases for a component or system under test, where the postcondition of one test is often used as the precondition for the next one.“ In Rahmen dieser Arbeit wird diese Definition übernommen, wobei die *Postcondition*[26] eines Testfalls nicht als *Precondition*[27] des nächsten Testfalls benutzt wird. Nach der Durchführung jedes Testfalls, muss das System in den Initialzustand zurück versetzt werden, damit alle Testfälle mit der gleichen Vorbedingung ausgeführt werden können. Eine Testsuite ist dann angemessen, wenn die *kumulierte Überdeckung* der beinhalteten Testfälle hoch und gleichzeitig die Anzahl der Testfälle so gering

[24]Beispiele für Testfälle werden unter anderem in Abschnitt 4.3.1 dargestellt

[25]Es wird eingestuft, ob sich die Software bei der Ausführung eines Testfalls richtig oder falsch verhalten hat

[26]Engl. für *Nachbedingung*: Bedingung, die für die Software nach Durchführung des Tests gilt

[27]Engl. für *Vorbedingung*: Bedingung, die für die Software vor Durchführung des Tests gilt

wie möglich ist.

Testen ist eine sehr zeitaufwändige und kostspielige Aktivität. Deshalb bedarf ein gründlicher und systematischer Test einer gut ausgearbeiteten Teststrategie, die schon in den frühen Phasen des Entwicklungsprozesses bestimmt wird. Es müssen qualifizierte Testexperten eingebunden werden, die mit der Testplanung gleich nach dem Start des Projekts anfangen. Zur Organisation des Testprozesses sind im Allgemeinen folgende Schritte vorgesehen:

1. *Testplanung*: In der Planungsphase werden die Testziele festgelegt und bestimmt wie diese zu erreichen sind. Unter anderem muss bestimmt werden, welches Personal eingesetzt wird, welche unterstützenden Werkzeuge benutzt werden und nach welchen Methoden vorgegangen wird. Auch das Testabschlusskriterium muss festgelegt werden, bevor alle Ergebnisse in einem Testplan festgehalten werden.
2. *Testfallgenerierung*: unter Berücksichtigung der Testziele und der durch die Testplanung vorgegebenen Randbedingungen sollen im Rahmen dieses Schrittes *ausführbare Testfälle* erstellt werden.
3. *Testdurchführung*: die im Rahmen der Testfallgenerierung erstellten Testfälle müssen in diesem Schritt manuell oder automatisch ausgeführt werden. Die Testdurchführung ist üblicherweise mit der Testauswertung stark verzahnt.
4. *Testauswertung*: Dieser Schritt vergleicht die bei der Ausführung der Testfälle gelieferten Ergebnisse mit den erwarteten Ergebnissen (diese werden anhand der Spezifikation bestimmt). Danach wird der durchgeführte Test im Hinblick auf die in der Testplanung vorgegebenen Ziele bewertet, speziell um festzustellen, ob das Testendekriterium erfüllt wurde. Darauf aufbauend wird ein Testbericht erstellt.

Die zeitaufwändigsten und damit auch kostspieligsten Schritte sind die *Testfallgenerierung* und die *Testdurchführung*. Deshalb gibt es viele Forschungsansätze und Werkzeuge, die diese Schritte unterstützen.

Ansätze zur *Automatisierung der Testdurchführung* sind in der Industrie etabliert [KTS10], wobei einer der bekanntesten Vertreter solcher Ansätze das *xUnit-Framework* ist. Zu diesem Framework gehören unter anderem das *JUnit*[28] Testing Framework für Java Programme und das *NUnit*[29] Framework für *.NET* Programme. Diese Frameworks erlauben es, eine Menge von

[28]http://www.junit.org/
[29]http://www.nunit.org/

Testskripten[30] manuell zu erstellen. Diese können dann automatisch ausgeführt werden, um für die einzelnen Testfälle die *Verdicts* automatisch zu ermitteln. Die Testskripte enthalten zum einen Aufrufe an die zu testenden Bausteine und zum anderen so genannte *Assertions*, die das gelieferte Ergebnis mit dem erwarteten Ergebnis automatisch zu vergleichen erlauben.

Als alternativer Ansatz zur Unterstützung der automatischen Testdurchführung ist noch der *Capture-Replay*-Ansatz zu erwähnen. Am Anfang ist auch bei diesem Ansatz manuelles Eingreifen nötig, um Testfälle zu erzeugen, die danach beliebig oft automatisiert durchgeführt werden können. Dazu wird zunächst die zu testende Software von einer Person bedient, während gleichzeitig die Benutzeraktionen (Mausklicks, Tastatureingaben usw.) zusammen mit den Ausgaben des Werkzeugs erfasst werden (*Capture*). Die so erfassten Testfälle können dann beliebig oft ausgeführt werden (*Replay*). Die Bewertung der gelieferten Ergebnisse bei erneuter Ausführung der Testfälle erfolgt durch Vergleich der Ausgabe des Werkzeugs mit der bei der ersten Ausführung gelieferten Ausgabe.

Ansätze zur *automatischen Testfallgenerierung* gibt es ebenfalls sehr viele, diese sind aber nicht so etabliert wie die zur Testdurchführung. Als Grundlage für die automatische Testfallgenerierung gilt dabei entweder der Quelltext der Software, wie z. B. in der Arbeit von [Ost07] oder Modelle, die das beabsichtigte Softwareverhalten beschreiben. Auf modellbasierte Ansätze wird in Abschnitt 3.2 und 3.3 eingegangen.

2.5.1. Testphasen

Abhängig von deren Art variiert der Zeitpunkt, zu dem sich Fehler manifestieren: so gibt es z. B. Fehler, die bereits beim Test einzelner Komponenten bemerkt werden. Andere Fehler äußern sich erst beim Zusammenspiel mehrerer Komponenten oder bei hoher Belastung der Software (z. B. durch viele gleichzeitige Aufrufe). Dementsprechend gibt es mehrere Testphasen, wobei jede auf die Auffindung spezieller Arten von Fehlern ausgerichtet ist.

2.5.1.1. Komponententest

Dieser wird auch *Unittest* oder *Modultest* genannt und bezeichnet den systematischen Test einzelner Softwarebausteine, der unmittelbar nach deren Implementierung durchgeführt werden

[30] Ein Testskript ist ein automatisch ausführbarer Testfall zusammen mit einem Testorakel

kann. Da Komponenten meistens von anderen Komponenten abhängig sind und/oder anderen Komponenten Funktionalitäten zur Verfügung stellen, muss eine geeignete *Testumgebung* hergestellt werden. Dies wird realisiert, in dem *Testtreiber* (Engl. *Driver*) und/oder *Platzhalter* (Engl. *Stub*) umgesetzt werden. Ein Testtreiber ist eine Komponente, deren einzige Funktionalität darin besteht, Dienste der zu testenden Komponente aufzurufen. Dagegen simuliert ein Platzhalter eine Komponente, die der getesteten Komponente Dienste zur Verfügung stellt. Meistens bedeutet die Programmierung eines Platzhalters mehr Aufwand als die Programmierung eines Testtreibers, da der Treiber nur Aufrufe enthalten muss; dagegen muss der Platzhalter eine bestimmte Funktionalität simulieren.

2.5.1.2. Integrationstest

Im Rahmen dieser Testart werden mehrere Komponenten zu einem Gesamtsystem zusammengesetzt mit dem Ziel, speziell deren Zusammenspiel zu testen. Es sollen dabei Fehler an den Schnittstellen der Komponenten aufgedeckt werden. Für das Zusammensetzen der Komponenten gibt es mehrere Strategien, die laut [Hof08] in drei Klassen aufgeteilt werden können:

Big-Bang-Integration

Diese Vorgehensweise als Strategie zu bezeichnen ist etwas unpassend, da sich dahinter keine wirklich systematische Herangehensweise verbirgt. Sie gibt nur vor, dass erst nach Entwicklung und Test aller benötigten Bausteine mit der Integration begonnen wird, indem alle Bausteine auf einmal zu einem Ganzen zusammengesetzt werden. Diese Vorgehensweise ist aber meistens nicht zu empfehlen, da sie bedeutende Nachteile mit sich bringt. Zum einen kann mit dem Integrationstest erst begonnen werden, nachdem alle Bausteine fertig gestellt worden sind; zum anderen wird – im Falle, dass Fehler entdeckt werden – die Suche nach der fehlerhaften Komponente erschwert.

Strukturorientierte Integration

Die Strategien dieser Klasse verlangen, dass das Gesamtsystem inkrementell aufgebaut wird: zunächst sollte nur ein Teil der Komponenten zu einem Subsystem zusammengesetzt und getestet werden. Dieses Subsystem wird dann nach und nach um weitere Komponenten erweitert. Welche Komponenten zuerst integriert werden, wird anhand der Architektur bestimmt.

Ein kleiner Nachteil dieser Strategien (im Vergleich zur Strategie Big-Bang-Integration) ist,

dass für die noch nicht integrierten Komponenten Testtreiber und/oder Platzhalter programmiert werden müssen, was zusätzlichen Aufwand bedeutet. Die Vorteile strukturorientierter Integration überwiegen aber wie die folgenden Ausführungen zeigen.

Top-Down-Integration ist eine Strategie die fordert, dass als erstes die Komponenten der obersten Softwareschicht, also die *anwendernahen Komponenten*[31], integriert werden. Dies sind meistens Komponenten, die andere Komponenten aufrufen, selbst aber nicht von anderen Komponenten, sondern nur von der Umgebung (z. B. durch den Benutzer) aufgerufen werden. Da die dadurch aufgerufenen Komponenten am Anfang nicht mit integriert werden, werden Platzhalter benötigt. Ein Vorteil dieser Strategie ist, dass die anwendernahen Komponenten früh verfügbar sind und getestet werden können. Somit kann der Kunde früh in den Beurteilungsprozess mit einbezogen werden.

Bottom-Up-Integration: entgegengesetzt zur Top-Down-Strategie verlangt diese, dass erst die *systemnahen Komponenten* integriert werden. Dies sind Komponenten, die meistens außer Betriebssystemfunktionen keine anderen Funktionen für die Umsetzung deren Funktionalität aufrufen. Im Falle der Anwendung dieser Strategie werden keine Platzhalter, sondern nur Testtreiber benötigt.

In der Praxis kommt es dann oft vor, dass Mischformen dieser Strategien zum Einsatz kommen; ein Beispiel dafür ist die *Outside-In-Integration*[32]. Sie versucht die Vorteile der Top-Down- und Bottom-Up-Strategien zu verbinden (sowohl anwendernahe als auch systemnahe Komponenten werden am Anfang integriert und getestet), indem erst die Module der obersten und der untersten Softwareschicht integriert werden. Danach werden die Komponenten der nächstniedrigen und der nächsthöheren Schicht hinzugezogen, bis letztendlich die Komponenten der mittleren Schicht mit betrachtet werden. Der Nachteil dabei ist, dass sowohl Testtreiber als auch Platzhalter programmiert werden müssen.

Sehr selten eingesetzt wird die Strategie *Inside-Out-Integration*. Durch den Anfang bei den Komponenten der mittleren Softwareschicht bietet sie nicht die Vorteile von Top-Down und Bottom-Up und hat die gleichen Nachteile wie Outside-In.

[31] Z. B. Benutzerschnittstellen-Komponenten

[32] Auch Sandwich-Strategie genannt

Funktionsorientierte Integration

Die Strategien dieser Klasse sind auch inkrementell; der Unterschied zu den gerade beschriebenen Strategien besteht darin, dass die Reihenfolge der Integration jetzt nicht mehr von der Architektur, sondern von funktionalen oder operativen Kriterien bestimmt wird.

Ein Repräsentant dieser Klasse ist die *Termingetriebene Integration* (Engl. *Schedule Driven*). Diese gibt vor, dass nach dem Prinzip *First Come First Serve*, Komponenten entsprechend ihrer Verfügbarkeit integriert werden sollen.

Die *Risikogetriebene Integration* (Engl. *Risc Driven*) geht nach dem Prinzip *Hardest First* vor; es werden erst die Komponenten zusammengesetzt, deren Ausfall ein hohes Risiko darstellt.

Die *Testgetriebene Integration* (Engl. *Test Driven*) orientiert sich an Testfällen und verlangt, dass erst einmal die Komponenten integriert werden müssen, die für die Ausführung bestimmter Testfälle benötigt werden.

Ähnlich wie die testgetriebene Integration geht auch die *anwendungsgetriebene Integration* (Engl. *Use Case Driven*) vor, mit dem Unterschied, dass hier nicht Testfälle als Startpunkt dienen, sondern einzelne Use-Cases[33] oder Geschäftsprozesse.

Es gibt also eine Fülle von Strategien. Die Wahl der im Einzelfall optimalen Integrationsteststrategie obliegt dem Testmanager, wobei dazu die Randbedingungen des Projekts zusammen mit dem Projektmanager analysiert werden müssen.

2.5.1.3. Systemtest

Diese Testart zielt auf die Bewertung der Funktionalität der Software hinsichtlich der Erfüllung der Kunden-Anforderungen. Der *Systemtest* ist sehr wichtig, weil er die letzte Möglichkeit bietet Fehler zu finden und zu beseitigen, bevor die Software an den Kunden ausgeliefert wird. Alle Fehler die danach gefunden werden, beeinflussen den Kunden direkt und verursachen neben Kundenunzufriedenheit einen hohen Aufwand bei der Behebung.

Die Testumgebung sollte jetzt der Produktionsumgebung beim Kunden möglichst ähnlich sein. Um Kosten zu senken wird diese Testphase oft in der Produktionsumgebung des Kunden selbst

[33]Siehe Abschnitt 2.2

durchgeführt, was ein Risiko darstellt, da in diesem Fall ein Ausfall der Software zu hohen Schäden auf Kundenseite führen kann.

Ähnlich wie im Rahmen des Integrationstests wird das Gesamtsystem als Menge von kollaborierenden Komponenten getestet. Der *große Unterschied zum Integrationstest* besteht allerdings darin, dass jetzt rein funktional getestet wird. Informationen über die Struktur der einzelnen Komponenten oder über deren Schnittstellen und Interaktionen sind hierbei irrelevant. Es soll allein die Benutzung durch den Kunden simuliert und überprüft werden, ob und in welchem Umfang die funktionalen und nicht-funktionalen Anforderungen erfüllt wurden. Die in [Bal97] beschrieben Systemtestarten sind:

- *Funktionstest*: im Rahmen dieser Systemtestart muss überprüft werden, ob die einzelnen Funktionalitäten *korrekt im Sinne des Auftraggebers* umgesetzt wurden.
- *Leistungstest*: der Leistungstest dient der Überprüfung des angestrebten Leistungsverhaltens[34]. Das System wird im Rahmen des Leistungstests hohen Belastungen ausgesetzt (indem es z. B. mit umfangreichen Datenmengen gefüttert wird), um zu überprüfen, wie es sich in solchen Fällen verhält und ob die umgesetzten Ausnahmebehandlungsmechanismen korrekt funktionieren.
- *Benutzbarkeitstest*: ob die Software leicht verständlich und intuitiv bedienbar ist, soll der Benutzbarkeitstest überprüfen.
- *Sicherheitstest*: ob Sicherheitsmaßnahmen umgesetzt wurden, ist im Rahmen dieser Testart zu untersuchen. Interessante Fragestellungen in diesem Fall sind z. B. wie die Zugriffsrechte verteilt sind, ob die Passwörter verschlüsselt gespeichert werden oder ob bestimmte Anforderungen an die Passwörter gestellt werden (z. B. Passwörter müssen eine Mindestlänge haben oder Passwörter müssen sowohl Buchstaben als auch Zahlen enthalten).
- *Interoperabilitätstest*: um die Software auf Kompatibilität mit der geplanten Hard- und Softwareumgebung zu testen, muss ein Interoperabilitätstest durchgeführt werden.
- *Konfigurationstest*: falls die Software für verschiedene Hardware-/Software-Plattformen geeignet ist, muss die Software auf allen möglichen Plattformen durch einen Konfigurationstest überprüft werden.
- *Dokumentenprüfung*: dies ist kein Test im eigentlichen Sinne. Im Rahmen dieser Aktivität

[34]In [Bal97] werden vier verschiedene Leistungstestausprägungen beschrieben: *Massentest*, *Zeittest*, *Lasttest*, *Streßtest*

soll überprüft werden, ob die Dokumentation für den Benutzer (in Form von Bedienungsanleitungen) und für die Entwickler (im Form von Entwurfsspezifikationen) ausreichend detailliert und korrekt vorhanden ist.

- *Installationstest*: Tests dieser Art überprüfen, ob sich die Software so installieren lässt, wie es im Benutzerhandbuch beschrieben ist.
- *Wiederinbetriebnahmetest*: es soll durch solche Tests geprüft werden, ob ein System nach dessen Ausfall wieder in Betrieb genommen werden kann.

2.5.1.4. Abnahmetest

Im Rahmen dieser Testart wird anhand der Vorgaben des Pflichtenhefts bewertet, in welchem Umfang die Software den Vertrag zwischen Kunden und Entwickler erfüllt. Dieser Test wird in der Einsatzumgebung des Kunden durchgeführt, mit dem Ziel, dass der Kunde am Ende dieser Testphase über die Abnahme oder Ablehnung des Systems entscheiden kann.

Aufgrund der Tatsache, dass während des Systemtests die Produktionsumgebung des Kunden nur simuliert wird, kann es beim Abnahmetest vorkommen, dass Testfälle, die in der simulierten Umgebung beim Entwickler als bestanden markiert wurden, bei Ausführung in der Kundenumgebung Fehler aufdecken.

2.5.1.5. Regressionstest

Die bisher beschriebenen Testphasen werden vor der Erstinstallation der Software beim Kunden durchgeführt. Der Lebenszyklus ist allerdings nicht mit der Abnahme durch den Kunden beendet. Es folgen noch die Phasen der *Wartung* und *Ablösung* der Software.

Im Rahmen der Wartung werden Änderungen an der Software vorgenommen. Dies kann mehrere Gründe haben, was dazu führt, dass es mehrere *Wartungsarten* gibt, die in [Som10] wie folgt klassifiziert werden:

- *Fault repairs*: falls beim Einsatz in der Kundenumgebung Fehler auftreten, müssen diese im Rahmen einer Fehlerkorrektur behoben werden. Dies wird auch als *Corrective Maintenance (fehlerbehebende Wartung)* bezeichnet.
- *Environmental adaption*: falls sich die Randbedingungen für den Einsatz der Software

beim Kunden ändern (z. B. ein unterschiedliches Betriebssystem), muss die Software daran angepasst werden. Ein anderer Begriff für diese Wartungsart ist *Adaptive Maintenance (anpassende Wartung)*.

- *Functionality addition*: Diese Wartungsart wird dann durchgeführt, wenn sich die Anforderungen des Kunden ändern oder der Kunde neue Funktionalitäten benötigt. Sie wird auch noch *Perfective Maintenance (vervollkommende Wartung)* genannt.

Nicht in der Klassifikation enthalten, aber trotzdem erwähnenswert ist die *Preventive Maintenance (vorbeugende Wartung)*. Diese Wartungsart hat das Ziel, die Software so zu ändern, dass spätere Wartungseingriffe jeglicher Art erleichtert werden. Dazu gehören z. B. die Aktualisierung der Dokumentation und Code-Refactorings.

In der Praxis kann nicht jede Änderung immer eindeutig einer bestimmten Wartungskategorie zugeordnet werden. So kann es z. B. vorkommen, dass bei der Anpassung an neue Randbedingungen zusätzliche Funktionalitäten hinzugefügt werden, um aus den neuen Randbedingungen den größtmöglichen Nutzen zu ziehen.

Alle Wartungsarten haben gemeinsam, dass dabei Änderungen in ein bestehendes und getestetes System eingefügt werden. Nach Durchführung der Änderungen muss das geänderte System erneut geprüft werden. Zum einen müssen die geänderten Codeteile überprüft werden und zum anderen muss aber auch sichergestellt werden, dass die Änderungen keine unerwünschten Seiteneffekte haben. Die hierzu eingesetzte Testmethode nennt sich *Regressionstest*. In [Lig09] wird das Ziel der Regressionstestphase wie folgt beschrieben:

> „Das Ziel des Regressionstests ist nachzuweisen, dass Modifikationen von Software keine unerwünschten Auswirkungen auf die Funktionalität besitzen.“

Im Rahmen des Regressionstests sollten idealerweise auch nach kleinsten Softwareänderungen, alle vorhandenen Testfälle noch einmal ausgeführt werden. Falls aufgrund bestimmter Projekteinschränkungen nicht alle Testfälle ausgeführt werden können, kann basierend auf den Informationen über die durchgeführten Änderungen ein so genannter *selektiver Regressionstest* (Engl. *Selective Regression Testing*) [FIMN07] durchgeführt werden. Das bedeutet, dass nicht alle Testfälle nach Änderungen wieder ausgeführt werden, sondern nur die, die dazu führen, dass geänderte Codeteile durchlaufen werden.

2.5.2. Testarten

Um Testfälle manuell oder automatisch abzuleiten, wird Zugang zu bestimmten Informationen über die zu testende Software benötigt. Diese Informationen können beispielsweise aus der textuellen Spezifikation, den Entwurfs-Modellen oder dem Quelltext erhoben werden. Anhand des Informationsstands, der der Generierung zugrunde gelegt wird, werden Testansätze in folgende Kategorien aufgeteilt.

2.5.2.1. Black-Box-Test

Diese Testart wird auch *funktionales Testen* oder *funktionsorientierter Test* genannt und wird wie folgt definiert:

Definition 2.3 (Black-Box-Test, aus [Ist10]) *Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.*

Mittels dieser Testart kann die Übereinstimmung eines Softwaresystems mit seiner Spezifikation überprüft werden. Das zu testende System wird hierbei als Black-Box betrachtet, was bedeutet, dass Informationen über die innere Funktionsweise der Software nicht von Bedeutung sind. Als Basis für die Testfallerstellung dient bei diesem Vorgehen allein die Spezifikation. Durch die Ausführung solcher Testfälle und durch Auswertung der Testergebnisse zeigt sich, inwiefern das Softwareprodukt der Spezifikation entspricht. Nachteil dieser Vorgehensweise ist, dass dadurch nicht abgeschätzt werden kann, zu welchem Anteil der Quelltext durch die Ausführung dieser Testfälle überdeckt wurde. Dies ist deshalb problematisch, weil sich in den möglicherweise unüberdeckten Quellcodeteilen Fehler befinden könnten.

Es gibt verschiedene Arten von Black-Box-Testverfahren. Die *funktionale Äquivalenzklassenbildung* beschreibt eine Systematik, die darauf basiert, dass der ganze Eingaberaum der Software in einzelne Klassen aufgeteilt wird, so dass alle Eingaben aus einer bestimmten Klasse von der Software gleichartig bearbeitet werden. Deshalb kann davon ausgegangen werden, dass jede Klasse durch eines ihrer Elemente adäquat repräsentiert wird. Darauf aufbauend verlangt dieses Testverfahren, dass die Software mit mindestens einem Repräsentanten jeder Äquivalenzklasse getestet wird.

Eine Erweiterung der funktionalen Äquivalenzklassenbildung stellt die *Grenzwertanalyse* dar, die als Eingaben nicht beliebige Vertreter der Äquivalenzklasse verlangt, sondern speziell die Vertreter, die sich an den Grenzen der Klassen befinden. Dieses Verfahren basiert auf der Erfahrung, dass Fehler in der Software besonders häufig auftreten, wenn Werte ausgewählt werden, die an den Grenzen der Äquivalenzklassen liegen [Lig09].

2.5.2.2. White-Box-Test

Diese Testart wird auch *Strukturtest*, *strukturelles Testen* oder *strukturorientierter Test* genannt und wird wie folgt definiert:

Definition 2.4 (White-Box-Test, aus [Ist10]) *Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.*

Wie es der Name schon suggeriert, orientieren sich die Verfahren, die sich dieser Kategorie zuordnen lassen, an der Struktur des zu testenden Programms. Bei der Testfallgenerierung werden Testfälle so erstellt, dass sie eine *möglichst hohe Überdeckung des Quelltextes* erreichen, ohne dabei die Spezifikation bei der Generierung in Betracht zu ziehen. Die Spezifikation ist allerdings auch in diesem Fall von entscheidender Bedeutung, und zwar bei der Bewertung der Ergebnisse der Testdurchführung.

Vorteil dieser Verfahren ist, dass der erstellte Quelltext viel gründlicher getestet wird, als bei einer rein funktionalen Herangehensweise. Problematisch ist allerdings, dass nicht abgeschätzt werden kann, ob das implementierte Verhalten das in der Spezifikation beschriebene Verhalten vollständig abdeckt.

Beispiele für Fehlerarten, die mit solchen Verfahren nicht entdeckt werden können, sind so genannte *Omission Faults*[35]. Andererseits können White-Box-Verfahren *unspezifiziertes Verhalten* aufdecken. White-Box-Testverfahren lassen sich in kontrollflussorientierte und datenflussorientierte Verfahren aufteilen.

[35] Fehler die daraus resultieren, dass in der Spezifikation beschriebene Funktionalität nicht vollständig oder gar nicht durch das implementierte Werkzeug umgesetzt wurde

Kontrollflussorientierter Test

Solche Testverfahren gehen von der Repräsentation des Quelltextes der Software als *Kontrollflussgraph* aus. In [Bal97] wird dieser Graph wie folgt definiert:

Definition 2.5 (Kontrollflussgraph) *Ein Kontrollflußgraph ist ein gerichteter Graph, der aus einer endlichen Menge von Knoten besteht. Jeder Kontrollflußgraph hat einen Startknoten und einen Endknoten. Die Knoten sind durch gerichtete Kanten verbunden. Jeder Knoten stellt eine ausführbare Anweisung dar. Eine gerichtete Kante von einem Knoten i zu einem Knoten j beschreibt einen möglichen Kontrollfluß vom Knoten i zum Knoten j. Die gerichteten Kanten werden als Zweige bezeichnet. Eine abwechselnde Folge von Knoten und Kanten, die mit dem Startknoten beginnt und mit einem Endknoten endet, heißt Pfad.*

In Anlehnung an [Bal97] werden im Folgenden kontrollflussorientierte Überdeckungskriterien beschrieben. Eines der bekanntesten Überdeckungskriterien, die auf die Überdeckung des Codes und damit auch des Kontrollflussgraphen ausgerichtet ist, ist die *Anweisungsüberdeckung*[36]. Dieses Kriterium verlangt die Überdeckung aller Anweisungen des Programms, d. h. aller Knoten im Kontrollflussgraphen. Durch Ausführung von Testfällen, die das Kriterium Anweisungsüberdeckung erfüllen, wird sichergestellt, dass im Programm alle Anweisungen mindestens einmal ausgeführt wurden. Falls für bestimmte Anweisungen keine geeigneten Testfälle gefunden werden können, ist dies ein Indiz dafür, dass es sich dabei um nicht ausführbare Anweisungen handelt. Laut einer in [Bal97] zitierten Studie, hat die Anweisungsüberdeckung – im Vergleich zu anderen kontrollflussorientierten Testverfahren – die geringste Fehleridentifizierungsquote. Diese geringe Quote ist darauf zurückzuführen, dass dieses Kriterium zwar die einzelnen Anweisungen betrachtet, nicht aber die verschiedenen Kontrollflüsse zwischen ihnen.

Aufgrund der geringen Leistungsfähigkeit der Anweisungsüberdeckung gilt die *Zweigüberdeckung*[37] als *das* minimale Testkriterium für White-Box-Verfahren. Dieses Kriterium *subsumiert*[38] die Anweisungsüberdeckung, da sie die Überdeckung aller Kanten des Kontrollflussgraphen fordert, was automatisch auch bedeutet, dass alle Knoten zu überdecken sind. Sie wird auch als *Entscheidungsüberdeckung* bezeichnet, da jede Entscheidung im Kontrollflussgraphen

[36] Auch C0-Test genannt

[37] Auch C1-Test genannt

[38] Ein Kriterium K subsumiert ein anderes Kriterium L, wenn jede beliebige Testfallmenge, die K erfüllt, auch L erfüllt

die Wahrheitswerte *wahr* und *falsch* mindestens einmal annehmen muss. Ihre Fehleridentifikationsquote ist besser als die der Anweisungsüberdeckung [Bal97].

Trotzdem bleiben viele Fehler durch Testfälle, die Zweigüberdeckung erreichen, unentdeckt. Dies ist darauf zurückzuführen, dass dieses Kriterium komplexe Pfade, die unter anderem auch Schleifen enthalten können, nicht adäquat berücksichtigt. Dasselbe trifft auch für komplexe, zusammenhängende Bedingungen zu.

Um besonders komplexe Bedingungen zu überprüfen, eignen sich *Bedingungsüberdeckungstests*. Von dieser Testart gibt es mehrere Ausprägungen, die zum einen die einzelnen atomaren Teilbedingungen[39] und zum anderen Variationen von atomaren Teilbedingungen betrachten.

Das schwächste Kriterium dieser Art ist die *einfache Bedingungsüberdeckung*. Dieses Kriterium fordert, dass Testfälle die Software so ausführen müssen, dass jede atomare Teilbedingung mindestens einmal zu *wahr* und einmal zu *falsch* ausgewertet wird. Da dieses Kriterium weder die Zweigüberdeckung noch die Anweisungsüberdeckung subsumiert, ist es für den praktischen Einsatz nicht ausreichend.

Deshalb wurde mit der *minimalen mehrfachen Bedingungsüberdeckung* ein anspruchsvolleres Kriterium eingeführt, welches verlangt, dass jede Bedingung – atomar oder nicht – mindestens einmal zu *wahr* und einmal zu *falsch* ausgewertet wird. Somit subsumiert dieses Kriterium die Zweigüberdeckung.

Das anspruchsvollste Kriterium dieser Art ist das Kriterium *mehrfache Bedingungsüberdeckung*. Dieses Kriterium besagt, dass alle Variationen atomarer Bedingungen zu überdecken sind. Da sich bei einer Bedingung mit n atomaren Teilbedingungen 2^n Variationsmöglichkeiten ergeben, ist dieses Kriterium im praktischen Einsatz meistens nicht erreichbar.

Um Schleifen zu testen eignet sich das *Pfadüberdeckungskriterium*. Es fordert, dass alle unterschiedlichen Anweisungssequenzen des Programms, also alle Pfade im Kontrollflussgraphen überdeckt werden. Dieses Kriterium nimmt unter den kontrollflussorientierten Testverfahren eine Sonderstellung ein: es ist viel anspruchsvoller als die Anweisungs- oder Zweigüberdeckung, aber meistens nicht sinnvoll einsetzbar, da in nicht-trivialen Programmen die Anzahl an Pfaden durch den Kontrollflussgraphen sehr hoch ist; falls Schleifen im Kontrollussgraphen vorkommen, kann die Anzahl an Pfaden sogar unbegrenzt sein.

[39]Sind Bedingungen die keine untergeordneten Bedingungen enthalten; z. B.: die zusammengesetzte Bedingung $(a > 0)$ && $(b == true)$ besteht aus den atomaren Teilbedingungen $(a > 0)$ und $(b == true)$

Deshalb gibt es abgeschwächte Varianten dieses Kriteriums, die die Anzahl der Schleifendurchläufe begrenzen, z. B. der *strukturierte Pfadtest*. Dabei wird ein maximaler Wert k für die Anzahl der Schleifendurchläufe vorgegeben und es wird die Überdeckung aller Pfade verlangt, die Schleifen höchstens k mal durchlaufen. Ein Spezialfall davon mit $k = 2$ ist der *Boundary-Interior-Pfadtest*.

Datenflussorientierter Test

Diese Testart gehört auch zu den Strukturtestverfahren. Im Gegensatz zu den kontrollflussorientierten Verfahren zielen diese Verfahren nicht auf die Überdeckung aller Knoten, Kanten oder Pfade des Kontrollflussgraphen, sondern auf die Überdeckung von Variablendefinitionen samt dazugehöriger Variablenverwendungen.

Um diese Kriterien zu beschreiben, müssen neben dem in Definition 2.5 eingeführten Kontrollflussgraphen noch weitere Begriffe eingeführt werden. Eine *Definition* (Kurz *Def*) einer Variable tritt in einer Anweisung des Programms auf (kann also einem Knoten im Kontrollflussgraphen zugeordnet werden), wenn der Variable bei der Durchführung dieser Anweisung ein Wert zugewiesen wird. Falls der Wert einer Variable im Rahmen einer Anweisung ausgewertet wird, dann handelt es sich um eine *Verwendung* der Variable (Engl. *Use*). Bei den Verwendungen von Variablen wird noch zwischen berechnender und prädikativer Verwendung unterschieden. Eine *berechnende Verwendung* (Engl. *Computational Use*, kurz *C-use*) kommt dann vor, wenn der Wert dieser Variable bei der Ausführung einer datenverarbeitenden Anweisung benutzt wird. Solch ein C-use kann also genau einem Knoten im Kontrollflussgraphen zugeordnet werden. Die *prädikative Verwendung* (Engl. *Predicative Use*, kurz *P-use*) einer Variable bedeutet, dass diese Variable bei der Ausführung einer Bedingungsauswertung innerhalb eines *Verzweigungsknotens*[40] ausgewertet wird, um den Wahrheitsgehalt dieser Bedingung zu bestimmen. Der Wahrheitsgehalt wird dann entscheiden, welche der ausgehenden Kanten durchlaufen wird. Ein P-use wird den Kanten zugeordnet, die von dem Verzweigungsknoten ausgehen.

Beispiel Die Anweisung $a := 0$ beschreibt eine Wertzuweisung der Variable a und enthält somit ein Def von a. Wenn dieser Wert in einer anderen Anweisung $b := a + 1$ benutzt wird, dann kommt hier sowohl eine C-use von a vor als auch ein Def von b vor. Eine spätere Bedingung über

[40] Dies ist ein Knoten mit zwei ausgehenden Kanten. Er repräsentiert eine Programmverzweigung (z. B. *if*) oder Schleifen (z. B. *for* oder *while*)

die Variable b: $if\ (b > 5)\ then$ enthält ein P-use von b.

Ein *datenflussannotierter Kontrollflussgraph* ist ein Kontrollflussgraph, der um Informationen zu *Defs* und *Uses* erweitert wurde.

Die Überdeckungsziele datenflussbasierter Kriterien beziehen sich auf Definitionen und Verwendungen gleicher Variablen. Es wird angestrebt, Paare von Definitionen einer Variable und dazugehörigen Verwendungen der Variable zu überdecken; genauer gesagt geht es darum, Pfade des datenflussannotierten Kontrollflussgraphen zwischen dem Knoten, der die Definition enthält und dem Knoten oder der Kante, der die Verwendung der Variable zugeordnet ist, zu überdecken, wobei entlang des Pfades keine erneute Definition der betrachteten Variable vorkommen darf. Solche Pfade nennt man *definitionsfreie Pfade* (Engl. *Def-clear-path*) bezüglich einer bestimmten Variable.

Basierend auf diesen Konzepten führten [RW85] datenflussbasierte Überdeckungskriterien ein. Diese werden hier kurz beschrieben, in Anlehnung an [Bal97]. Diese Kriterien sind – was den Aufwand und die Fehlererkennbarkeit angeht – zwischen den einfachen kontrollflussbasierten Kriterien wie Zweigüberdeckung und den sehr komplexen und im Allgemeinen nicht erfüllbaren Pfadüberdeckungskriterien angesiedelt.

Da zwischen einer Definition und einer Verwendung entlang des Kontrollflussgraphen im Allgemeinen mehrere definitionsfreie Pfade existieren, genügt es im Falle der meisten unten beschriebenen Kriterien (mit Ausnahme von *All-DU-paths*), wenn mindestens einer dieser Pfade überdeckt wird.

Das *All-defs*-Kriterium verlangt, dass jede Kombination einer Variablendefinition mit mindestens einer dazugehörigen Verwendung getestet wird. Falls keine Verwendung der entsprechenden Variable gefunden werden kann, ist die Variable und die entsprechende Zuweisung im Quelltext sinnlos. Als schwächstes Kriterium dieser Klasse subsumiert es weder das Kriterium Anweisungsüberdeckung noch das Kriterium Zweigüberdeckung.

Ein Kriterium, das diese zwei Kriterien subsumiert, ist das *All-p-uses*-Kriterium. Dieses Kriterium gilt dann als erfüllt, wenn jede Variablendefinition in Kombination mit jeder dazugehörigen prädikativen Verwendung getestet wird.

Für den Fall, dass solche Paare nicht existieren, wurde das *All-p-uses/some-c-uses*-Kriterium definiert. Dieses fordert zunächst ähnlich wie das All-p-uses-Kriterium die Überdeckung aller

Paare von Definitionen und entsprechenden prädikativen Verwendungen; für den Fall, dass für bestimmte Variablen keine prädikativen Verwendungen existieren, soll zumindest eine berechnende Verwendung dieser Variablen überdeckt werden. Somit subsumiert dieses Kriterium das All-p-uses-Kriterium.

Das *All-c-uses*-Kriterium fordert, dass alle Variablendefinitionen in Kombination mit allen dazugehörigen berechnenden Verwendungen getestet werden. Zusätzlich dazu fordert das *All-c-uses/some-p-uses*-Kriterium, dass – falls keine berechnende Verwendung einer Variable existiert – zumindest eine prädikative Verwendung erreicht wird. Dieses Kriterium subsumiert in der Regel nicht die Zweigüberdeckung; es subsumiert allerdings das All-defs- und das All-c-uses-Kriterium.

Ein weiteres Kriterium, das alle bisher beschriebenen Datenflusskriterien subsumiert, ist das *All-uses*-Kriterium. Dieses fordert, dass alle Definitionen in Kombination mit allen dazugehörigen prädikativen und berechnenden Verwendungen getestet werden. Wie schon vorhin erwähnt, gibt es ein Kriterium, für dessen Erfüllung es nicht genügt, einen einzigen Pfad zwischen Definition und entsprechender Verwendung zu überdecken. Da die Überdeckung aller möglichen Pfade genau so unrealistisch ist wie im Falle des kontrollflussbasierten Pfadüberdeckungskriteriums, beschränkt sich das *All-DU-Paths*-Kriterium auf die Überdeckung aller *schleifenfreien Pfade* zwischen allen Definitionen und allen erreichbaren Verwendungen.

2.5.2.3. Grey-Box-Test

Grey-Box-Testverfahren stellen eine Kombination zwischen White-Box- und Black-Box-Testverfahren dar. Solche Testverfahren konzentrieren sich auf die Testfallgenerierung anhand von Informationen, die nicht so detailliert wie der Software-Quelltext sind, dafür aber im Vergleich zur Spezifikation der Software Umsetzungsdetails enthalten. Durch Kenntnis bestimmter Umsetzungsdetails kann die Software etwas genauer getestet werden als nur anhand der Spezifikation wie es im Black-Box-Test der Fall ist. Da im Grey-Box-Test nicht die gesamte Quelltextkomplexität zu überdecken ist, ist diese Testart im Vergleich zu White-Box-Testverfahren mit deutlich weniger Aufwand umsetzbar. Somit stellen Grey-Box-Verfahren einen guten Kompromiss zwischen der Genauigkeit der White-Box- und dem vergleichsweise geringem Aufwand von Black-Box-Testverfahren dar.

2.6. Modellbasierter Test

Modellbasiertes Testen ist eine Testart, die formale Modelle benutzt, um daraus Testfälle nach vorgegebenen Testkriterien weitgehend automatisch zu generieren. Da Modelle auf einer Abstraktionsebene zwischen Code und textueller Spezifikation anzusiedeln sind, stellt das modellbasierte Testen eine Ausprägung von Grey-Box-Verfahren dar. In [UPL06] wird *modellbasiertes Testen* (Engl. *Model-Based Testing*, kurz *MBT*) wie folgt definiert:

Definition 2.6 (modellbasierter Test) *A variant of testing that relies on explicit behaviour models that encode the intended behaviour of a system and possibly the behaviour of its environment.*

Laut [RBGW10] gibt es verschiedene Ausprägungen von modellbasierten Testverfahren:

- *modellorientiertes Testen*: bei diesem Vorgehen werden Modelle nur als Leitfaden benutzt. Sie dienen als Grundlage für die Diskussion zwischen Projektbeteiligten und zum Verständnis der Domäne. Sie dienen also nicht explizit der Testfallgenerierung.
- *modellgetriebenes Testen*: Modelle werden hier nicht mehr nur als Leitfaden gesehen; das Ziel dieses Ansatzes ist es, ausführbare Testfälle aus diesen Modellen zu generieren. Dazu bedarf es vor allem leistungsfähiger Modellierungswerkzeuge und Werkzeuge zur Testfallgenerierung.
- *modellzentrisches Testen*: im Rahmen dieses Vorgehens wird die modellbasierte Testfallgenerierung nicht mehr als „Einbahnstraße“ betrachtet: zusätzlich zur Generierung von Testfällen aus Modellen, sollen bei diesem Vorgehen alle relevanten Informationen aus der Testdurchführung in das der Testfallgenerierung zugrunde liegende Modell mit einfließen.

Gemäß dieser Klassifizierung lässt sich der in dieser Arbeit beschriebene Testfallgenerierungsansatz der Kategorie *modellgetriebenes Testen* zuordnen.

Eine weitere Systematik MBT-Ansätze zu kategorisieren wird in [UPL06] vorgeschlagen; zusammengefasst wird diese Kategorisierung in Abbildung 2.2. Hier werden sieben voneinander abhängige[41] Dimensionen mit verschiedenen Ausprägungen vorgeschlagen. Diese Dimensionen lassen sich in drei Hauptgruppen gliedern: *Modellierung* (*Model*), *Testgenerierungsmethode*

[41] So z. B. hat die Auswahl eines bestimmten Modellparadigmas (Paradigm) entscheidenden Einfluss auf das einsetzbare Testfallgenerierungsverfahren (Technology)

(*Test Generation*) und *Durchführungstechnologie* (*Test Execution*). Die vertikalen Pfeile mit zwei Spitzen (kommen bei den Dimensionen *Subject* und *Redundancy* vor) deuten auf eine kontinuierliche Reihe von Möglichkeiten hin. Das bedeutet, dass auch Mischformen der dargestellten Alternativen möglich sind. Die geschwungenen Linien zeigen Alternativen, die sich nicht unbedingt gegenseitig ausschließen. Dagegen kennzeichnet das Symbol „/“ an den Blättern dieses Baumes sich gegenseitig ausschließende Alternativen.

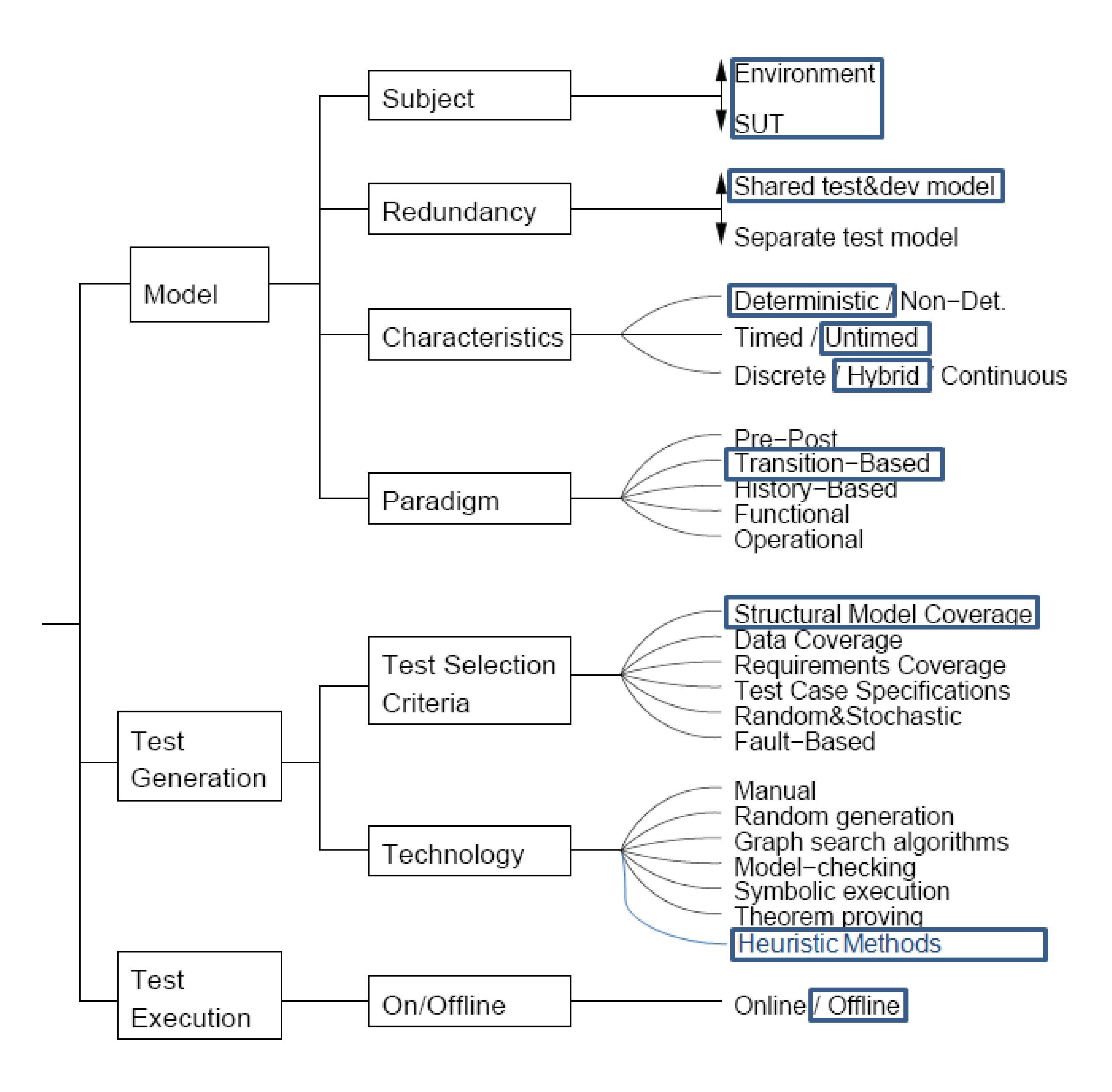


Abbildung 2.2.: Taxonomie modellbasierter Testansätze (aus [UPL06])

Einzelne Begriffe aus der Abbildung 2.2 wurden vom Bearbeiter dieser Arbeit durch Umrandung gekennzeichnet, um zu zeigen, wie sich der im Rahmen dieser Arbeit beschriebene modellbasierte Testfallgenerierungsansatz in diese Taxonomie einordnen lässt. Bei der Dimension *Test Generation: Technology* wurde ein neuer Begriff eingefügt, da in dieser Taxonomie die im Rahmen dieser Arbeit beschriebene Methode zur Testfallgenerierung nicht erwähnt wird, und zwar *genetische Algorithmen*[42], die zur Klasse der *heuristischen Verfahren*[43] (Engl. *Heuristic Methods*) zählen. Im Folgenden werden die einzelnen Dimensionen der Taxonomie dargestellt und speziell die Konzepte erklärt, die für die Einordnung der vorliegenden Arbeit relevant sind. Für eine detaillierte Beschreibung der Taxonomie wird auf [UPL06] oder [UL07] verwiesen.

1. Die Dimension *Subject* gibt an, ob das Modell das Softwaresystem an sich (*SUT: System Under Test*) oder die Außenwelt (*Environment*) beschreibt, die mit dem System interagiert. Die als Basis für die Testfallgenerierung in dieser Arbeit benutzten Modelle sind Modelle, die reaktive Softwaresysteme beschreiben. Somit beschreiben sie das SUT, welches auf Eingaben der Umwelt reagiert, was dazu führt, dass auch ein Teil des Außenweltverhaltens in diesen Modellen enthalten ist. Dies wird durch Umrandung beider Begriffe gekennzeichnet.

2. Die Dimension *Redundancy* gibt an, ob ein *Entwicklermodell* (*Shared test&dev model*) oder ein *separates Testmodell* (*Separate test model*) der Testfallerstellung zugrunde liegt.

   Bei den Entwicklermodellen handelt es sich um Modelle, die im Zuge der modellbasierten Softwareentwicklung während der konstruktiven Entwurfsphasen entstanden sind. Auf der anderen Seite kann für die Generierung der Testfälle ein dediziertes, separates Testmodell erstellt werden. Dieses fasst mehrere Testfälle als Modell in einer übersichtlichen Darstellung zusammen, was dazu führt, dass dieses Vorgehen besonders im Vergleich zu der manuellen Testfallerstellung viele Vorteile bietet.

   Die Benutzung der Entwicklermodelle bei der Testfallgenerierung bietet gegenüber der Benutzung separater Testmodelle den Vorteil, dass für die Testfallgenerierung keine zusätzlichen Modelle erstellt werden müssen, was geringere Kosten bedeutet. Darüber hinaus liefert die Überdeckung der Entwicklermodelle beim Test einen Indikator dafür, inwiefern die gesamte modellierte Funktionalität überprüft wurde.

   Ein Kritikpunkt hinsichtlich der Benutzung der Entwicklermodelle zur Testfallgenerierung

[42] Siehe Kapitel 4
[43] Siehe [DS06]

ist es, dass beim Testen eines Programmcodes, der automatisch aus Entwicklermodellen generiert wurde, nur die automatische Transformation von Modell in Code überprüft wird. Für Domänen in denen die Erstellung von Code zu einem hohen Anteil automatisiert ist (z. B. im Bereich der Codegenerierung für die *Electronic Control Units* im Automotive-Bereich), mag diese Befürchtung auch berechtigt sein. Bei den meisten entwickelten kommerziellen Softwaresystemen ist allerdings immer noch sehr viel manueller Programmieraufwand nötig, da sich die automatische Generierung in vielen Fällen auf die Generierung von statischem Code (z. B. Java-Klassengerüsten[44]) beschränkt. Das Verhalten dieser Klassen (Implementierung der Methoden) muss dagegen *manuell programmiert werden*. In diesem Fall macht die Generierung von Testfällen anhand von Entwicklermodellen sehr viel Sinn, weil dadurch überprüft werden kann, ob die gewünschte und modellierte Funktionalität auch tatsächlich implementiert wurde.

Ein Nachteil der Testfallgenerierung aus Entwicklermodellen besteht darin, dass durch dieses Vorgehen keine Omission Faults entdeckt werden können: falls das Modell nicht alle Anforderungen des Kunden berücksichtigt, können solche Fehler mit Hilfe von Testfällen, die aus dem Modell generiert wurden, nicht entdeckt werden. Es besteht also ein ähnliches Problem wie bei der Testdurchführung von White-Box-Testfällen[45].

Die Benutzung separater Testmodelle führt eine zusätzliche Perspektive (die Perspektive des Testers) auf das Verhalten der Software ein und erlaubt es deshalb auch speziell die vorhin erwähnten Omission Faults zu erkennen. Andererseits kann durch Überdeckung des Testmodells nicht abgeschätzt werden, ob das gesamte umgesetzte Verhalten durch die generierten Testfälle überprüft wurde.

3. Eine weitere Dimension bezieht sich auf die *Charakteristiken des Modells* (*Characteristics*). Dabei wird unterschieden zwischen deterministischen und nicht-deterministischen Modellen. Des Weiteren stellt sich die Frage, ob Zeit bei der Modellierung eine Rolle spielt und ob es sich um ein diskretes, kontinuierliches oder um ein hybrides Modell handelt. Diskrete Systeme sind dynamische Systeme, deren Zustand sich durch das Auftreten von Ereignissen ändert. Kontinuierliche Systeme ändern ihren Zustand, ohne dafür auf das Auftreten externer Ereignisse angewiesen zu sein. Hybride Systeme sind Mischformen dieser zwei Varianten.

[44] Diese enthalten Klassenname, Attribute und Methodensignaturen

[45] Siehe 2.5.2

Die im Rahmen dieser Arbeit betrachteten Modelle enthalten deterministische, hybride Zustandsautomaten, welche keine zeitlichen Bedingungen beinhalten.

4. Die vierte Dimension ist das *Modellparadigma* (*Paradigm*). Dieses bezieht sich auf die Art der verwendeten Modellierungssprache: dabei kann es sich unter anderem um *transitionsbasierte Systeme* handeln (wie z. B. die in dieser Arbeit betrachteten Zustandsautomaten) oder um Systeme, deren Operationen durch *Preconditions* und *Postconditions* definiert werden; in diesem Fall handelt es sich um *State-Based Notations*. Ein Beispiel für so eine Sprache ist die Sprache Z [Spi89].

5. *Überdeckungskriterien* (*Test Selection Criteria*) stellen eine weitere Dimension dieser Taxonomie dar. Ähnlich wie auf Codeebene gibt es speziell für *transitionsbasierte Systeme* eine ganze Reihe von Überdeckungskriterien. Da die zur Modellierung solcher Systeme am häufigsten benutzten Diagramme (z. B. Zustandsautomaten oder Aktivitätsdiagramme) Graphen sind, kann ein großer Teil der in Abschnitt 2.5.2 beschriebenen White-Box-Testkriterien[46] auf Modelle übertragen werden.

   Die im Rahmen dieser Arbeit betrachteten Kriterien lassen sich der Kategorie *Structural Model Coverage* zuordnen und werden in Abschnitt 3.2 beschrieben.

6. Die sechste Dimension heißt *Technology* und bezieht sich auf die Vorgehensweise bei der Testfallgenerierung. Zum einen wird die manuelle Generierung erwähnt; da aber speziell Modelle eine sehr gute Basis für die automatische Testfallgenerierung darstellen, werden hauptsächlich automatische Vorgehensweisen beschrieben. Dabei gibt es die Möglichkeit Testfälle zufällig zu generieren: *Random Generation*. Diesem unsystematischen Verfahren stehen viele systematische Verfahren, wie z. B. *Graphensuche* oder *Model Checking* gegenüber.

7. Die letzte Dimension der Taxonomie ist *Test Execution*. Hier geht es um die zeitliche Abfolge der Testfallgenerierung und Durchführung. Wird die Testsuite erst vollständig generiert, um danach ausgeführt zu werden, handelt es sich um ein *Offline*-Verfahren. Als Alternative dazu kann bei der Generierung die Ausgabe des SUT mit berücksichtigt werden, was einem *Online*-Verfahren entspricht [Utt05]. Bei dem im Rahmen dieser Arbeit beschriebenen Verfahren, handelt es sich um ein Offline-Verfahren.

[46] Diese sind auch auf die Überdeckung eines Graphen ausgerichtet, genauer gesagt auf die Überdeckung des Kontrollflussgraphen

Unabhängig von dem ausgewählten Testfallgenerierungsansatz können mit modellbasiert generierten Testfällen unterschiedliche Ziele verfolgt werden. Hier sind zwei Szenarien möglich:

- *Szenario 1*: in diesem Fall werden die generierten Testfälle auf Modellebene ausgeführt, mit dem Zweck *Modellierungsfehler* zu finden. Solche Fehler schon während der Modellierung und nicht erst später im Entwicklungsprozess zu entdecken, ist sehr wichtig, da dadurch die Auswirkungen der Fehler minimiert werden.

  Mit dieser Zielsetzung beschäftigt sich z. B. der in [AFGC03] beschriebene Ansatz. Hier geht es darum, aus Klassendiagrammen und Kommunikationsdiagrammen Testfälle zu generieren, um die Diagramme an sich zu überprüfen.

- *Szenario 2*: andererseits können solche Testfälle benutzt werden, um zu überprüfen, ob die Implementierung dem im Modell beschriebenen Verhalten entspricht. In diesem Fall wird vorausgesetzt, dass das Modell korrekt ist und Testfälle werden dazu benutzt, um das Verhalten der Implementierung gegenüber dem vom Modell beschriebenen Verhalten zu prüfen. Diese Vorgehensweise ermöglicht das Finden von *Implementierungsfehlern*.

  Dabei ist der gewählte Grad der Modellabstraktion von entscheidender Bedeutung: Testfallmengen, die eine bestimmte strukturelle Überdeckung eines Modells erreichen, welches sehr detailliert die zu implementierende Funktionalität beschreibt, haben eine höhere Chance Implementierungsfehler zu entdecken als Testfallmengen, die eine gleiche strukturelle Überdeckung auf einem abstrakteren Modell erreichen.

  Die meisten Ansätze zum modellbasierten Test, die im Rahmen dieser Arbeit genannt werden (z. B. [Sok04], [WS07]), verfolgen dieses Ziel.

Diesen Szenarien wird auch bei der Evaluierung in Kapitel 7 Rechnung getragen, indem das Potential modellbasierter Testfälle sowohl hinsichtlich der Erkennung von Modellierungsfehlern als auch hinsichtlich der Erkennung von Implementierungsfehlern untersucht wird.

3. Modellbasierte Überdeckungskriterien

Bevor in Kapitel 4 die zur Testfallgenerierung eingesetzten genetischen Algorithmen vorgestellt werden, beschreibt dieses Kapitel im ersten Abschnitt die der Testfallgenerierung zu Grunde liegende Modellnotation. Danach werden modellbasierte Überdeckungskriterien beschrieben: zum einen werden in Abschnitt 3.2 Überdeckungskriterien für den Komponententest vorgestellt; zum anderen werden Überdeckungskriterien für den Integrationstest in Abschnitt 3.3 vorgestellt, gefolgt von der Beschreibung zustandsbasierter Integrationstestkriterien in Unterabschnitt 3.3.2.

3.1. Verhaltensbeschreibung von Komponenten mittels UML-Zustandsautomaten

Als Sprache zur Modellierung des Verhaltens einzelner Komponenten wurden Zustandsautomaten ausgewählt. Sie gehören zu den Verhaltensdiagrammen der UML und sind eine der meist benutzten Diagrammarten. Da Zustandsautomaten sehr mächtig sind und sehr viele Sprachkonstrukte zur Verfügung stellen, würde eine detaillierte Beschreibung all dieser Konstrukte den Rahmen der Arbeit sprengen. Deshalb werden nur die Modellierungskonstrukte beschrieben, die im Rahmen dieser Arbeit eingesetzt wurden.

Die UML-Spezifikation beschreibt zwei Arten von Zustandsautomaten, und zwar *Verhaltenszustandsautomaten* (Engl. Behavioral State Machine) und *Protokollzustandsautomaten* (Engl. Protocol State Machine). Diese unterscheiden sich unter anderem in der Semantik von Transitionen; so kann z. B. für eine Transition in einem Protokollzustandsautomaten kein Verhalten angegeben werden, während dies für Verhaltensautomaten möglich ist (das Verhalten kann in Effects von Transitionen beschrieben werden[1]). Da sie geeignete Sprachkonstrukte zur Modellierung des Verhaltens an Transitionen zur Verfügung stellen, werden im Folgenden nur Verhaltenszustands-

[1] Wie das genau passiert wird in Unterabschnitt 3.1.2 beschrieben

automaten betrachtet. Wenn im Rahmen dieser Arbeit also Zustandsautomaten erwähnt werden, sind immer Verhaltenszustandsautomaten gemeint.

3.1.1. Zustand

Ein für Zustandsautomaten zentrales Konzept, ist der *Zustand* (Engl. *State*). Die Interpretation des Zustands nach [OMG10] ist:

> „A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some behavior (i.e., the model element under consideration enters the state when the behavior commences and leaves it as soon as the behavior is completed)."

Der Zustand beschreibt also eine Situation, in der sich eine Komponente oder ein Softwaresystem befinden kann, wobei während dieser Situation bestimmte Invarianten erfüllt sind. Es kann sich dabei um eine statische Situation (z. B. das System zeigt ein Dialogfeld an und wartet auf Eingaben durch den Benutzer) oder um eine dynamische Situation (z. B. das System führt Berechnungen durch) handeln. Zu einem bestimmten Zeitpunkt befindet sich das System in genau einem Zustand, dem *aktiven Zustand*. Dargestellt werden Zustände als Rechtecke mit abgerundeten Ecken.

Eine spezielle Art von Zuständen sind die so genannten *Pseudozustände*. Diese dienen unter anderem zur einfachen Darstellung komplexer Beziehungen zwischen Zuständen. Beispiele für Pseudozustände sind Verzweigungen oder History-Zustände. Ein weiterer Pseudozustand ist der Startzustand[2], der den Zustand angibt, der beim Betreten des Zustandsautomaten aktiv wird.

3.1.2. Transition

Die durchgezogenen und gerichteten Pfeile zwischen Zuständen jeglicher Art, stellen Transitionen dar und beschreiben Übergänge zwischen den Zuständen. Nachdem eine – aus dem aktiven Zustand ausgehende – Transition durchlaufen wurde, wird dieser Zustand inaktiv und der Zielzustand der Transition aktiv. Die Übergänge zwischen Zuständen erfolgen ohne zeitliche Verzö-

[2]Engl. Initial Pseudo State, wird dargestellt als schwarzer gefüllter Kreis

gerung. Mittels Beschriftungen werden die Bedingungen erfasst, die gelten müssen, damit eine Transition durchlaufen wird. Eine solche Beschriftung hat die Form [RHQ+07]:

$$Trigger[Guard]/Effect,$$

wobei alle Elemente optional sind. Aufgrund dieser Beschriftung und der Tatsache, dass eine Transition einen Quell- (Engl. Pre-state) und einen Zielzustand (Engl. Post-state) hat, kann eine Transition als 5-Tupel folgendermaßen dargestellt werden [SOP07]:

$$t = (pre(t),\ tr(t),\ g(t),\ e(t),\ post(t)).$$

Im Folgenden werden die Elemente des Tupels in Anlehnung an [SOP07] und [RHQ+07] erklärt, wobei zur Veranschaulichung auf die in Abbildung 3.1 dargestellte Komponente A hingewiesen wird. Hier ist an jeder Transition über der Beschriftung auch der Name der Transition angebracht (*tA1* bis *tA8*).

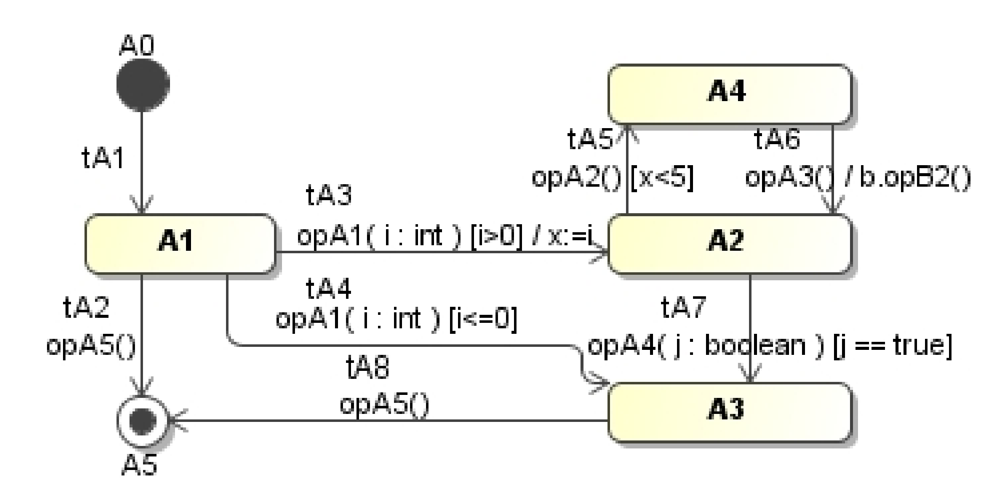


Abbildung 3.1.: Zustandsautomat, der das Verhalten einer Komponente *A* beschreibt (aus [PSO08])

$pre(t)$: *Pre-state* der Transition *t*, bezeichnet den Zustand einer Komponente, der aktiv war, bevor *t* durchlaufen wurde.

$tr(t)$: *Trigger* der Transition *t*, bezeichnet ein Ereignis, das *t* auslösen kann. Ob die Transition *t* dann durchlaufen wird, hängt auch davon ab, ob der Guard (siehe unten) erfüllt ist. Der Trigger kann auch formale Parameter haben. Generell muss eine Transition nicht unbedingt einen Trigger enthalten. Falls die Transition keinen Trigger hat, wird sie sofort durchlaufen wenn ihr Pre-state aktiv wird und der Guard erfüllt ist.

$tr(t) = [_]$ bezeichnet das Fehlen eines Triggers an der Transition t.

$g(t)$: *Guard* der Transition t, bezeichnet eine Bedingung, die erfüllt sein muss, damit t durchlaufen wird; Die Bedingung kann dabei über:

- die Parameter des Triggers formuliert werden (z. B.: Guard $[i > 0]$ bezieht sich auf den Parameter i von $opA1(i : int)$) oder
- über die Variablen der umschließenden Komponente (z. B.: Guard $[x < 5]$ bezieht sich auf die Variable x der Komponente A).

$g(t) = [_]$ bezeichnet das Fehlen eines Guards an der Transition t.

Um mittels der in Kapitel 4 beschriebenen Methode Testfälle aus Zustandsautomaten zu generieren, ist es wichtig, dass die Guards der Transitionen, die aus dem selben Zustand ausgehen und denselben Trigger haben, untereinander disjunkt sind, so dass die Zustandsmaschinen deterministisches Verhalten beschreiben.

$e(t)$: *Effect* der Transition t, bezeichnet ein Verhalten, das beim Durchlaufen von t durchgeführt wird.

$e(t) = [_]$ bezeichnet das Fehlen eines Effects an einer Transition. Das im Effect beschriebene Verhalten kann beinhalten [SOP07]:

- *Variablenänderung*: Variablen, die im Kontext der aktuellen Komponente sichtbar sind, können im Rahmen einer Zuweisung verändert werden. Dabei können die Werte der Parameter des Triggers oder die Attribute der umschließenden Komponente referenziert werden. Beispiel für eine solche Zuweisung ist $x := i$ der Transition *tA3*. In diesem Fall wird der Variable x der Wert des Parameters i zugewiesen.

  Sei $EV_C := \{e(t) | t \in T_C \wedge e(t)\ besteht\ nur\ aus\ Variablenzuweisungen\}$ die Menge aller Effects einer Komponente C, die nur aus Variablenzuweisungen bestehen.

- *Aufrufe anderer Komponenten*: Neben der Variablenänderung kann das Verhalten auch Aufrufe anderer Komponenten beinhalten. So z. B. ist an der Transition *tA6* im Effect ein Aufruf der Komponente B modelliert.

  Sei $EA_C := \{e(t) | t \in T_C \wedge e(t)\ besteht\ nur\ aus\ Aufrufen\ anderer\ Komponenten\}$ die Menge aller Effects einer Komponente C, die nur aus Aufrufen anderer Komponenten bestehen.

Sei $EVA_C := \{e(t)|t \in T_C \wedge e(t)\ beinhaltet\ mindestens\ eine\ Variablenzuweisung\ und\ mindestens\ einen\ Aufruf\ einer\ anderen\ Komponente\}$ die Menge aller Effects einer Komponente C, die sowohl Variablenzuweisungen als auch Aufrufe anderer Komponenten enthalten. Diese Effects werden *generische Effects* genannt.

$post(t)$: *Post-state* der Transition t, bezeichnet den Zustand einer Komponente, der aktiv wird, nachdem t durchlaufen wurde.

Detailliert, als 5-Tupel aufgelistet werden die Transitionen des Zustandsautomaten aus Abbildung 3.1 in Tabelle 3.1.

Name	pre(t)	tr(t)	g(t)	e(t)	post(t)
tA1	A0 (Startzustand)	[_]	[_]	[_]	A1
tA2	A1	opA5()	[_]	[_]	A5 (Endzustand)
tA3	A1	opA1(i:int)	i>0	x:=i	A2
tA4	A1	opA1(i:int)	i<=0	[_]	A3
tA5	A2	opA2()	x<5	[_]	A4
tA6	A4	opA3()	[_]	b.opB2()	A2
tA7	A2	opA4(j:boolean)	j==true	[_]	A3
tA8	A3	opA5()	[_]	[_]	A5 (Endzustand)

Tabelle 3.1.: Detaillierte Beschreibung der Transitionen der Komponente *A*

Aufbauend auf diesen Konzepten werden nun die für das Durchlaufen einer Transition notwendigen Bedingungen beschrieben. Zu einem bestimmten Zeitpunkt befindet sich der Zustandsautomat in einem aktiven Zustand. Falls ein Aufruf an die Komponente geschickt wird, muss folgendes erfüllt sein, damit dieser Aufruf zu einem Zustandsübergang führt: aus dem aktiven Zustand muss eine Transition t ausgehen, deren Trigger gleich dem Aufruf ist und deren Guard erfüllt ist[3]. Wenn diese Bedingungen erfüllt sind, wird die Transition t durchlaufen und der neue aktive Zustand des Zustandsautomaten wird der Post-state (post(t)) der durchlaufenen Transition. Falls es keine Transition gibt, die diese beiden Bedingungen erfüllt, verfällt dieser Aufruf. Dies bedeutet, dass der Zustandsautomat im selben Zustand verbleibt, in dem er vor dem Eintreffen des Aufrufs war.

[3]Falls die Transition keinen Guard hat, also falls $g(t) = [_]$, dann gilt der Guard immer als erfüllt

Im weiteren Verlauf dieses Kapitels werden Überdeckungskriterien für den Komponenten- und Integrationstest vorgestellt. Zu jedem Kriterium, das im Rahmen dieser Arbeit bei der Testfallgenerierung eingesetzt wurde, wird eine Formel angegeben, die es erlaubt die Anzahl der zu überdeckenden Entitäten zu berechnen. Zunächst müssen deshalb an dieser Stelle ein paar zusätzliche Konzepte eingeführt werden. Manche Konzepte werden erst später bei der Beschreibung der Modell-Mutationsoperatoren in Abschnitt 7.1.2.2 benötigt.

Sei $COMP$ die Menge aller Komponenten eines Softwaresystems; für jede Komponente $C \in COMP$ werden im Folgenden ein Reihe von Begriffen beschrieben, von denen ein Großteil in [SOP07] eingeführt wurden[4]:

- S_C: die Menge der Zustände von C

  Beispiel: $S_A = \{A0,\ A1,\ A2,\ A3,\ A4,\ A5\}$

- T_C: die Menge der Transitionen von C

  Beispiel: $T_A = \{tA1,\ tA2,\ tA3,\ tA4,\ tA5,\ tA6,\ tA7,\ tA8\}$

- Für einen Zustand $S \in S_C$ seien die Mengen der damit verbundenen Kanten wie folgt definiert:
  - $Ek_S := \{t \in T_C | post(t) = S\}$: Menge der in Zustand S eingehenden Transitionen

    Beispiel: Die Menge aller eingehenden Transitionen des Zustands $A2 \in S_A$ ist: $Ek_{A2} = \{tA3,\ tA6\}$

  - $Ak_S := \{t \in T_C | pre(t) = S\}$: Menge der aus Zustand S ausgehenden Transitionen

    Beispiel: Die Menge aller ausgehenden Transitionen des Zustands $A2 \in S_A$ ist: $Ak_{A2} = \{tA5,\ tA7\}$

- $TP_C := \{(t_1,\ t_2) \in (T_C\ \times\ T_C) | \exists S \in S_C : t_1 \in Ek_S \wedge t_2 \in Ak_S\}$: die Menge aller Transitionspaare von C

  Beispiel: $TP_A = \{(tA1,\ tA2),\ (tA1,\ tA3),\ (tA1,\ tA4),\ (tA3,\ tA5),$

[4]Zur beispielhaften Beschreibung dieser Konzepte wird die Komponente A aus Abbildung 3.1 benutzt

$(tA3,\ tA7),\ (tA4,\ tA8),\ (tA5,\ tA6),\ (tA6,\ tA5),$

$(tA6,\ tA7),\ (tA7,\ tA8)\}$

- $Anfangszustand_C := \{S \in S_C \| |Ek_S| = 0\}$: der Anfangszustand von C

  Beispiel: $Anfangszustand_A = \{A0\}$

- $Endzustände_C := \{S \in S_C \| |Ak_S| = 0\}$: die Menge der Endzustände von C

  Beispiel: $Endzustände_A = \{A5\}$

- $Tr_C := \{tr(t) | t \in T_C\}$: die Menge aller Trigger von C

  Beispiel: $Tr_A = \{[_],\ opA1(i : int),\ opA2(),\ opA3(),$
  $opA4(j : boolean),\ opA5()\}$

- $G_C := \{g(t) | t \in T_C \wedge g(t) \neq [_]\}$: die Menge aller nicht trivialen Guards von C

  Beispiel: $G_A = \{i > 0,\ i <= 0,\ x < 5,\ j == true\}$

- $E_C := \{e(t) | t \in T_C \wedge (e(t) \in EA_C \vee e(t) \in EVA_C)\}$: die Menge aller Effects von C, die Aufrufe enthalten

  Beispiel: $E_A = \{b.opB2()\}$

- $T[E]_C := \{t \in T_C | e(t) \in E_C\}$: die Menge aller Transitionen, die im Effect einen Aufruf enthalten (die so genannten *Invoking Transitions* (*aufrufende Transitionen*))

  Beispiel: $T[E]_A = \{tA6\}$

- $T[Tr]_C(x) := \{t \in T_C | tr(t) = x\}$: die Menge aller Transitionen mit einem bestimmten Trigger x. Anhand des Triggers $opA1(i : int)$ wird diese Menge beispielhaft beschrieben:

  Beispiel: $T[Tr]_A(opA1(i : int)) = \{tA3,\ tA4\}$

- $Tr[E]_C(x) := \{tr \in Tr_C | \exists t \in T_C : tr(t) = tr \wedge e(t) = x\}$: die Menge aller Trigger von Transitionen mit einem Effect x. Anhand des Effects $b.opB2()$ wird diese Menge beispielhaft beschrieben:

  Beispiel: $Tr[E]_A(b.opB2()) = \{opA3()\}$

- $S[E]_C := \{s \in S_C | \exists t \in T_C : s = pre(t) \wedge e(t) \in E_C\}$: die Menge der Pre-states von *aufrufenden Transitionen*

  Beispiel: $S[E]_A = \{A4\}$

- $S[E]_C(x) := \{s \in S_C | \exists t \in T_C : s = pre(t) \wedge e(t) = x\}$: die Menge der Pre-states von Transitionen mit einem bestimmten Effect x. Anhand des Effects $b.opB2()$ wird diese Menge beispielhaft beschrieben:

  Beispiel: $S[E]_A(b.opB2()) = \{A4\}$

- $S[Tr]_C(x) := \{s \in S_C | \exists t \in T_C : s = pre(t) \wedge tr(t) = x\}$: die Menge der Pre-states von Transitionen mit einem bestimmten Trigger x. Anhand des Triggers $opA5()$ wird diese Menge beispielhaft beschrieben:

  Beispiel: $S[Tr]_A(opA5()) = \{A1,\ A3\}$

- $S[Tr,E]_C(x) := \{s \in S_C | \exists t \in T[E]_C : s = pre(t) \wedge tr(t) = x\}$: die Menge der Pre-states von *aufrufenden Transitionen* mit einem bestimmten Trigger x. Anhand des Triggers $opA3()$ wird diese Menge beispielhaft beschrieben:

  Beispiel: $S[Tr,E]_A(opA3()) = \{A4\}$

- $S[Tr,E]_C(x,y) := \{s \in S_C | \exists t \in T_C : s = pre(t) \wedge tr(t) = x \wedge e(t) = y\}$: die Menge der Pre-states von Transitionen mit einem bestimmten Trigger x und einem Effect y. Anhand des Triggers $opA3()$ und des Effects $b.opB2()$ wird diese Menge beispielhaft beschrieben:

  Beispiel: $S[Tr,E]_A(opA3(),\ b.opB2()) = \{A4\}$

In den nächsten zwei Abschnitten werden die bei der Testfallgenerierung betrachteten Kriterien vorgestellt. Für jedes Kriterium wird mit der Bezeichnung $\#\,NameKriterium$ die Anzahl zu überdeckender Entitäten angegeben. Hierbei werden auch Entitäten mitgezählt, die u.U. gar nicht überdeckt werden können. Da darüber hinaus mit einem Testfall mehrere Entitäten überdeckt werden können, beschreiben diese Metriken obere Schranken der zur Erfüllung entsprechender Kriterien benötigten Testfallanzahl.

3.2. Modellbasierte Überdeckungskriterien für den Komponententest

Ähnlich wie auf Codeebene gibt es auch für Zustandsautomaten eine ganze Reihe von Überdeckungskriterien. Da Zustandsautomaten Graphen sind, kann ein großer Teil der in Abschnitt 2.5.2 beschriebenen White-Box-Testkriterien auf Zustandsautomaten übertragen werden. Die am meisten verbreiteten Kriterien sind die, die auf die Überdeckung der Knoten und Kanten eines Zustandsautomaten ausgerichtet sind:

- *Zustandsüberdeckung*: verlangt die Überdeckung aller Zustände $S \in S_C$ des Zustandsautomaten einer Komponente C.

$$\#\,Zustandsüberdeckung := |S_C|$$

- *Transitionsüberdeckung*: verlangt, dass jede Transition $t \in T_C$ eines Zustandsautomaten durchlaufen wird. Ähnlich wie die Zweigüberdeckung auf Codeebene gilt die Transitionsüberdeckung als das minimale Testkriterium bei der Generierung von Testfällen aus Zustandsautomaten. Dass dies sinnvoll ist, wurde auch im Rahmen der Evaluierung verschiedener Überdeckungskriterien festgestellt[5], da die Testfälle für Transitionsüberdeckung deutlich mehr Fehler erkannten als Testfälle für Zustandsüberdeckung.

$$\#\,Transitionsüberdeckung := |T_C|$$

- *Transitionspaarüberdeckung*: verlangt, dass für jeden Zustand $S \in S_C$ der Zustandsmaschine einer Komponente C, jede eingehende Transition aus Ek_S in Kombination mit jeder aus-

[5] Siehe Abschnitt 7.1.3

gehenden Transition aus Ak_S (also jedes *Transitionspaar* des Zustandsautomaten) getestet wird.

$$\#\,Transitionspaarüberdeckung := \sum_{S \in S_C} |Ek_S| * |Ak_S| = |TP_C|$$

- *Pfadüberdeckung*: dieses Kriterium verlangt die Überdeckung aller Pfade durch den Zustandsautomaten. Ähnlich wie auf Codeebene ist dieses Kriterium allerdings im Allgemeinen nicht erfüllbar. Deshalb existieren auch abgeschwächte Formen dieses Kriteriums, z. B. das in [RBGW10] beschriebene *All-One-Loop-Paths*-Kriterium, welches die Überdeckung aller schleifenfreien Pfade und die Überdeckung der Pfade mit einem einmaligen Schleifendurchlauf verlangt.

Neben den kontrollflussbasierten Codeüberdeckungskriterien können auch die in 2.5.2 beschriebenen *Bedingungsüberdeckungskriterien* auf Zustandsautomaten übertragen werden. Solche Kriterien sind auf die Überdeckung der Bedingungen in den Guards[6] der Transitionen ausgerichtet. In [WS08] werden eine Reihe solcher Kriterien beschrieben. So z. B. verlangt das *Condition Coverage*-Kriterium, dass jede atomare Bedingung eines Guards einmal zu true und einmal zu false ausgewertet wird. Darüber hinaus werden in dieser Arbeit neue Kriterien definiert, die auf Bedingungsüberdeckungskriterien aufbauen und zusätzlich die Grenzen der Wertebereiche der Variablen in den Guards betrachten.

Auch die in Abschnitt 2.5.2 vorgestellten *Datenflusskriterien* können für Zustandsautomaten angepasst werden. Datenfluss wird in Zustandsautomaten wie folgt beschrieben: Variablen werden in Effects der Transitionen definiert, um danach in Guards oder anderen Variablenzuweisungen referenziert zu werden. Die Arbeit von [KHC+99] zielt auf die Überdeckung der Datenflüsse in Zustandsautomaten.

Alle bisher beschriebenen Ansätze zielen auf die Überdeckung des modellierten und somit gewünschten Verhaltens, das mittels Automaten beschrieben ist. Um zu überprüfen, wie sich das System verhält, wenn Aufrufe ankommen, die nicht dem modellierten Verhalten entsprechen, wird in [BH08] ein Verfahren eingeführt, das neben dem gewünschten Verhalten auch das nicht erwünschte Verhalten betrachtet.

[6] Siehe Abschnitt 3.1

Das Modell wird dazu um eine komplementäre Sicht erweitert. Dies bedeutet, dass im Automaten zum einen ein *Error State* und zum anderen mehrere *Faulty Transitions* wie folgt beschrieben eingeführt werden: für jeden *Simple State* (kein Error State) und für jedes Event des Automaten (welches nicht als Auslöser für eine Transition aus dem aktuell betrachteten Zustand gilt), wird eine Faulty Transition aus dem aktuellen Zustand in den Error State eingeführt. Somit beinhaltet der so genannte *Completed Statechart* neben den normalen Zuständen und Transitionen (den Simple States und Legal Transitions) sowohl einen Error State als auch Faulty Transitions.

Durch Fokussierung auf die Überdeckung der normalen Transitionen wird getestet, wie sich das System bei erwarteter Beanspruchung verhält. So verlangt z. B. das Kriterium *K-transition Coverage*, die Überdeckung aller Sequenzen von normalen Transitionen der Länge $k \in N$. Im Gegensatz dazu, verlangt das Kriterium *Faulty Transition Pair Coverage*, dass jedes Transitionspaar bestehend aus einer Legal Transition und einer Faulty Transition überdeckt wird.

3.3. Modellbasierte Überdeckungskriterien für den Integrationstest

Wie schon in Abschnitt 2.5.1 erwähnt, zielt der Integrationstest auf das Testen des Zusammenspiels mehrerer Komponenten ab. Die Durchführung dieser Testphase setzt voraus, dass die einzelnen interagierenden Komponenten bereits erfolgreich einen Komponententest bestanden haben, damit sichergestellt ist, dass die Komponenten ihre Spezifikation erfüllen. Falls dies nicht der Fall ist, wird die Suche nach der Ursache von Fehlern, die sich beim Integrationstest offenbaren, aufwendiger, denn der Fehler kann sowohl in der Interaktion der Komponenten untereinander liegen als auch in der inneren Funktionalität einzelner Komponenten.

Als erstes muss bei der Planung dieser Testphase entschieden werden, in welcher Reihenfolge die einzelnen Komponenten zusammengesetzt und getestet (z. B. Top-down oder Bottom-up[7]) werden. Danach müssen geeignete Testfälle generiert werden. Dabei stellt sich die Frage nach der besten Strategie für die Herleitung von Integrationstestfällen.

Eine reine Black-Box-Betrachtung des Gesamtsystems ist nicht detailliert genug, da eine solche Sicht zwar die Gesamtfunktionalität des Systems beschreibt, aber keinerlei Information über die einzelnen Komponenten und ihre Interaktionen enthält.

[7] Siehe Abschnitt 2.5.1

Auf der anderen Seite bietet eine White-Box-Perspektive auf die Komponenten Zugang zu diesen Informationen. Deshalb gibt es auf Quellcodeebene Überdeckungskriterien für den Integrationstest, die auf die Überdeckung der Interaktionen ausgerichtet sind, so z. B. die *Coupling Based Criteria*, die in [JO98] vorgeschlagen werden. Diese basieren auf den klassischen Datenflusskriterien[8] und betrachten die Konzepte der Def und Use von Variablen komponentenübergreifend. Diese Kriterien verlangen, dass Ausführungspfade zwischen Definitionen von Variablen in einer Komponente und ihren Verwendungen in anderen Komponenten überdeckt werden.

Für den Fall, dass die interagierenden Komponenten in einer objektorientierten Sprache programmiert wurden, muss der Integrationstest für solche Komponenten auch die spezifischen Konzepte dieser Sprache berücksichtigen. Aus diesem Grund betrachten die Kriterien von [AO04] die Konzepte der *Polymorphie* und *Vererbung*. Sie ziehen in Betracht, dass abhängig vom gegenwärtigen Kontext verschiedene polymorphe Methoden aufgerufen werden können, wobei sich in jeder der überschriebenen Methoden ein Use einer Variable befinden kann. Die entsprechende Def muss in Kombination mit den Uses aus den polymorphen Methoden überdeckt werden.

Neben der Frage nach einer adäquaten Überdeckung der implementierten Komponenteninteraktion, ist eine weitere interessante Fragestellung, die nach der quantitativen Zuverlässigkeitsbewertung eines Systems, das aus mehreren interagierenden Komponenten besteht. Zu diesem Zweck müssen ebenfalls Testfälle generiert werden, deren Auswertung die Ableitung statistischer Aussagen über das zu testende System ermöglicht. Mit der Generierung solcher Testfälle beschäftigt sich die Arbeit [MS11], die gleichzeitig das Betriebsprofil des Systems und die Komponenteninteraktionsüberdeckung (gemäß ausgewählter Überdeckungskriterien aus [JO98] und [AO04]) auf Codeebene berücksichtigt.

Die Anwendung von White-Box-Verfahren bei der Integrationstestfallgenerierung ist allerdings häufig nicht möglich, da der Quelltext der interagierenden Komponenten oft nicht vorliegt (dies ist besonders bei Off-the-shelf-Komponenten der Fall). Da die Black-Box-Perspektive ebenfalls nicht adäquat ist, wurde die Integrationstestfallgenerierung im Rahmen dieser Arbeit ausgehend von einer Grey-Box-Sicht auf die Komponenten durchgeführt. Es wurden UML-Modelle als Eingabe benutzt, die das Softwaresystem beschreiben. Dabei stellt die UML verschiedene Diagramme zur Verfügung, die sich für die Modellierung interagierender Komponenten eignen. Besonders Interaktionsdiagramme[9] sind speziell für die Beschreibung der Kompo-

[8] Siehe Abschnitt 2.5.2
[9] Siehe Abschnitt 2.2

nenteninteraktion konzipiert und dienen deshalb in vielen Forschungsarbeiten zum modellbasierten Integrationstest als Modellierungssprache und Grundlage für die Testfallgenerierung.

Die Arbeit von [RJBP05] beschreibt Überdeckungskriterien für den Integrationstest, die auf die Überdeckung von Komponentenschnittstellen ausgerichtet sind. Dabei wird nicht explizit angegeben, aus welchen Artefakten (Code oder Modell) diese Informationen erhoben werden. Die Kriterienbeschreibung basiert auf folgenden Definitionen: als *Ereignisse* (Engl. *Events*) werden Aufrufe von Schnittstellenoperationen definiert. Einzelne Events können abhängig voneinander sein, wobei zwischen zwei Arten von Abhängigkeiten unterschieden wird. *Context-dependence* bedeutet, dass ein Event als Folge des Aufrufs eines anderen Events aufgerufen wird. Zwei Events sind dagegen *Content-dependent*, wenn der Wert einer Variable durch ein Event geändert und durch das andere Event ausgelesen wird. Folgende Überdeckungskriterien wurden eingeführt: das *Interface Test Coverage*-Kriterium verlangt, dass jede Operation der Schnittstelle einer Komponente mindestens einmal aufgerufen wird; im Gegensatz dazu verlangt das *Event Test Coverage*-Kriterium, dass die Aufrufe der Operation aus jedem möglichen Kontext heraus überdeckt werden. Das *Context dependence Test Coverage*-Kriterium verlangt, dass Sequenzen von Events überdeckt werden, die in einer Context-dependent-Abhängigkeitsbeziehung stehen. Das *Content dependence Test Coverage*-Kriterium fordert, die Überdeckung von Events, die in einer Content-dependent-Abhängigkeitsbeziehung stehen.

Eine Reihe von Arbeiten zur modellbasierten Herleitung von Integrationstestfällen beschäftigen sich mit der automatischen Testfallgenerierung aus Sequenzdiagrammen: [RKS05], [SM10], [CNM07], [DT07] und [BB00]. Dabei sind die Kriterien darauf ausgerichtet, entweder *einzelne Messages* des Diagramms oder *Message-Sequenzen* zu überdecken. Ein Kriterium, das auf die Überdeckung einzelner Messages ausgerichtet ist, ist das in [DT07] beschriebene *All Message Coverage*-Kriterium, das wie folgt formuliert wurde:

> „Testing must cause each message in a sequence diagram to be sent at least once“.

Ein Beispiel für ein Überdeckungskriterium auf Sequenzdiagrammen, das Message-Sequenzen berücksichtigt ist das in [SM10] eingeführte *Message Sequence Path*-Kriterium:

> „For each sequence diagram, there must be at least one test case T such that when the software is executed using T, the software that implements the message sequence path of the sequence diagram must be executed.“

Semantisch äquivalent zum Sequenzdiagramm ist das Kommunikationsdiagramm. Auf die

Testfallgenerierung aus Kommunikationsdiagrammen konzentrieren sich eine Reihe von Arbeiten, wie z. B. [AO00], [WCO03] und [AFGC03]. Aufgrund der semantischen Äquivalenz zu Sequenzdiagrammen sind die für Kommunikationsdiagramme beschriebenen Überdeckungskriterien ähnlich wie die vorhin erwähnten Kriterien, die für Sequenzdiagramme definiert wurden. Als Beispiel für ein solches Kriterium sei das in [WCO03] definierte Kriterium:

> „Each valid sequence in each collaboration diagram has to be tested at least once.“[10]

Der Tatsache, dass in Kommunikationsdiagrammen auch Guards an den Messages modelliert werden können, wird in [AFGC03] Rechnung getragen. Dieser Ansatz beschreibt Überdeckungskriterien auf Kommunikationsdiagrammen, die neben der Überdeckung einzelner Messages (Kriterium: *Each message on link*) oder Message-Pfaden (Kriterium: *All message paths*) explizit auf die Überdeckung der Guards an den Messages ausgerichtet sind. So z. B. wird das Kriterium *Full predicate coverage* wie folgt beschrieben:

> „Given a test set T and collaboration diagram CD, T must cause each clause in every condition in CD to take the values of TRUE and FALSE while all other clauses in the predicate (condition) have values such that the value of the predicate will always be the same as the clause being tested.“

Die bisher betrachteten Ansätze definieren Überdeckungskriterien für einzelne Interaktionsdiagramme und fokussieren deshalb die Überdeckung einzelner Aufrufe oder Aufrufsequenzen zwischen verschiedenen Komponenten. Da die interagierenden Komponenten aus verschiedenen Zuständen heraus die im Sequenz- oder Kommunikationsdiagramm beschriebenen Aufrufe triggern können, verlangt der Ansatz von [Sok04], dass zusätzlich zu dem Sequenzdiagramm auch Zustandsautomaten benutzt werden sollen, um die kommunizierenden Objekte zu initialisieren. Durch unterschiedliche Initialisierung der einzelnen Objekte können mehrere Testfälle aus einem einzelnen Sequenzdiagramm generiert werden. Dieser Ansatz gibt aber keine expliziten Kriterien an, die eine automatische Generierung von entsprechenden Testfällen ermöglichen könnte.

Ein Schritt weiter in Richtung der Betrachtung der Zustände interagierender Komponenten wird in [ABuR+06] gegangen. Hier werden in einem ersten Schritt Zustandsautomaten und Kommunikationsdiagramme kombiniert, um ein neues Diagramm, das so genannte *State Collaboration Test Model* (Kurz *SCOTEM*) zu generieren. Auf Basis dieses Diagramms können

[10] In der Definition wird der Begriff *Collaboration Diagram* benutzt. Dies kommt daher, da das Kommunikationsdiagramm in früheren UML-Versionen Kollaborationsdiagramm (Engl. Collaboration Diagram) hieß

Testsequenzen für den Integrationstest automatisiert generiert werden. Dazu werden Überdeckungskriterien betrachtet, die auf die Überdeckung von Pfaden im SCOTEM ausgerichtet sind. Durch Überdeckung solcher Pfade werden die Zustände der *aufgerufenen* Komponenten betrachtet, d. h. es wird getestet, wie eine Komponente auf einen Aufruf abhängig von dem Zustand, in dem sie sich befindet, reagiert. Die beschriebene Vorgehensweise betrachtet allerdings nicht die *aufrufende* Komponente, also nicht die Tatsache, dass der Aufruf aus unterschiedlichen Zuständen bzw. aus unterschiedlichen Transitionen der *aufrufenden* Komponente erfolgen kann.

Zusammenfassend werden die aufgezählten modellbasierten Ansätze in Tabelle 3.2 dargestellt. Einzelne Ansätze betrachten neben den vorhin erwähnten Interaktionsdiagrammen und Zustandsautomaten auch andere Diagrammarten, die aber für die Integrationstestgenerierung nicht von Bedeutung sind. So z. B. betrachtet [AFGC03] auch Klassendiagramme, um die statische Struktur der Modelle zu überprüfen. Diese Diagrammarten werden in der Tabelle nicht berücksichtigt.

	S	K	Z
[RKS05]	x		
[SM10]	x		
[CNM07]	x		
[DT07]	x		
[BB00]	x		
[Sok04]	x		x
[AO00]		x	
[AFGC03]		x	
[WCO03]		x	
[ABuR$^+$06]		x	x

S: Sequenzdiagramm
K: Kommunikationsdiagramm
Z: Zustandsautomat

Tabelle 3.2.: Zusammenfassung existierender Ansätze, die modellbasierte Überdeckungskriterien für den Integrationstest beschreiben

Um sowohl die Zustände *der aufgerufenen als auch der aufrufenden Komponente* zu beachten,

wurden in [SOP07] neuartige *zustandsbasierte Überdeckungskriterien* definiert, welche in Abschnitt 3.3.2 beschrieben werden. Das Verfahren zur automatischen Generierung von Testfällen, die diese Kriterien erfüllen, wird in Kapitel 4 vorgestellt.

3.3.1. Mapping

Bevor im nächsten Abschnitt die zustandsbasierten Überdeckungskriterien vorgestellt werden, beschreibt dieser Abschnitt ein zentrales Konzept dieser Arbeit, und zwar das *Mapping*. Dieses Konzept wurde in [SOP07] eingeführt und ist wie folgt definiert:

Definition 3.1 (Mapping) *Ein Mapping ist ein Paar* (t, t') *von Transitionen unterschiedlicher Zustandsautomaten mit folgender Eigenschaft:*

e(t) = tr(t').

Ein Mapping ist also ein Paar von Transitionen, für das gilt, dass der Effect der *aufrufenden Transition t* gleich dem Trigger der *aufgerufenen Transition* t' ist. Diese Gleichheit ist so zu verstehen, dass im Effect von t ein Aufruf enthalten ist, der gleich mit dem Trigger von t' ist. Diese Gleichheit bedeutet also nicht, dass die Zeichenketten, die im Effect von t und im Trigger von t' modelliert sind, gleich sein müssen[11].

Für ein Softwaresystem, das aus einer Menge *COMP* von interagierenden Komponenten besteht, wird die Menge aller Mappings zwischen den interagierenden Komponenten in [SOP07] wie folgt definiert:

$$Map := \{(t, t') | \exists C, C' \in COMP : C \neq C' \wedge t \in T_C \wedge t' \in T_{C'} \wedge e(t) = tr(t')\}$$

Für jedes Mapping $m = (t, t') \in Map$ werden in [SOP07] noch folgende Konzepte beschrieben:

- $k(m) \in COMP$ die (*aufrufende Komponente*): also die Komponente C, die Transition t enthält und
- $k'(m) \in COMP$ die (*aufgerufene Komponente*): also die Komponente C', die Transition t' enthält.

[11] Es könnten im Effect von t neben dem Aufruf auch z. B. Variablenzuweisungen enthalten sein

Zur beispielhaften Beschreibung dieses Konzepts stellt Abbildung 3.2, neben den Zustandsautomaten der Komponente A auch den Zustandsautomaten einer weiteren Komponente B dar. Dabei enthält die Komponente A eine Transition *tA6*, deren Effect einen Aufruf der Komponente B enthält. Auf der anderen Seite hat Komponente B zwei Transitionen (*tB3* und *tB4*), deren Trigger gleich dem Effect von *tA6* ist. Zwischen diesen zwei Komponenten gibt es also zwei Mappings $m1 = (tA6,\ tB4)$ und $m2 = (tA6,\ tB3)$.

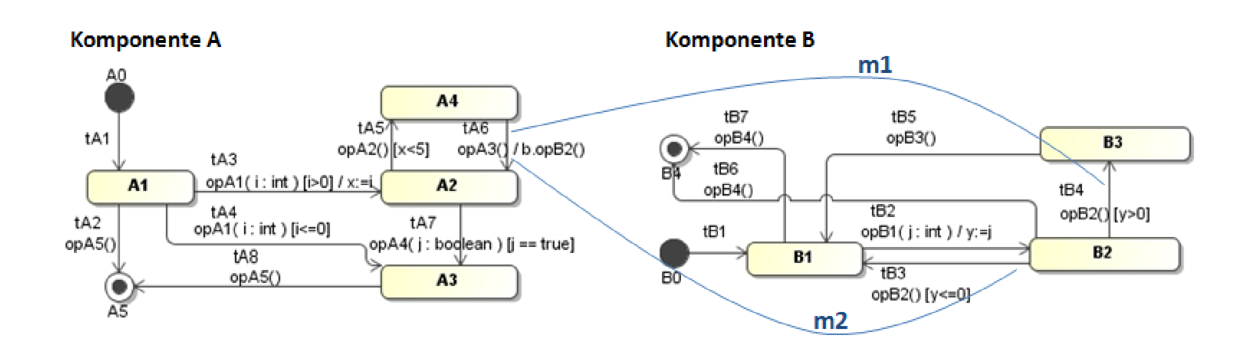


Abbildung 3.2.: Beispiel für ein Modell mit zwei interagierenden Komponenten (aus [PSO08])

Ein Mapping $m = (t,\ t')$ wird durch einen Testfall überdeckt, wenn bei der Ausführung des Testfalls zu einem bestimmten Zeitpunkt die aufrufende Transition t durchlaufen wird, was dazu führt, dass Komponente $k'(m)$ aufgerufen wird. Des Weiteren muss gelten, dass der Aufruf der Komponente $k'(m)$ dazu führt, dass die aufgerufene Transition t' durchlaufen wird, und zwar direkt nach dem t durchlaufen wurde[12]. Falls solch ein Testfall nicht konstruiert werden kann, ist das Mapping *nicht ausführbar*.

Neben dem Mapping-Konzept spielt auch das Konzept der *Mapping-Gruppe* eine wichtige Rolle. Diese ist eine Menge von zueinander alternativen Mappings[13]. Eine Mapping-Gruppe zu überdecken bedeutet zumindest ein Mapping dieser Gruppe zu überdecken. Falls eine Mapping-Gruppe ausschließlich aus nicht ausführbaren Mappings besteht, dann ist sie *nicht ausführbar*.

[12] D. h., dass zwischen dem Durchlaufen von t und t' keine anderen Transitionen irgendeiner Komponente des Systems durchlaufen wurden

[13] Zueinander alternative Mappings sind Mappings, von denen bei der Testfallgenerierung nur ein Mapping überdeckt werden muss

3.3.2. Zustandsbasierte Überdeckungskriterien

Da Transitionen als 5-Tupel dargestellt werden können und Mappings Paare von Transitionen sind, kann durch unterschiedlich feine Überdeckungsgrade der beiden 5-Tupel von t und t' eine Hierarchie zustandsbasierter Integrationstestkriterien definiert werden. Diese Kriterien werden zusammengefasst in Tabelle 3.3 dargestellt. Das „+“-Zeichen gibt zu jedem Kriterium an, welche Entitäten in Kombination betrachtet werden. Jedes Kriterium gibt vor, dass für Kombinationen von Entitäten Mappings überdeckt werden müssen, die bestimmte Bedingungen erfüllen. Zu beachten ist dabei, dass im Allgemeinen nicht zu jeder Kombination entsprechende Mappings existieren werden. Die Kombinationen, für die es keine entsprechenden Mappings gibt, werden von den Kriterien nicht betrachtet.

Kriterium	aufrufende Komp. C				aufgerufene Komp. C'	
	pre	tr	$e \in E_C$	$post$	pre	$post$
$Pre-states\ and\ triggers$	+	+	-	-	-	-
$Pre-states,\ triggers\ and\ effects$	+	+	+	-	-	-
$Invoking\ transitions$	+	+	+	+	-	-
$Effects\ on\ pre-states$	-	-	+	-	+	-
$Effects\ in\ /\ on\ pre-states$	+	-	+	-	+	-
$Triggers\ and\ effects\ in\ /\ on\ pre-states$	+	+	+	-	+	-
$Invoking\ transitions\ on\ pre-states$	+	+	+	+	+	-
$Invoking\ /\ invoked\ transitions$	+	+	+	+	+	+

Tabelle 3.3.: Zustandsbasierte Überdeckungskriterien für den Integrationstest (aus [SOP07])

Für zwei interagierende Komponenten $C,\ C' \in COMP$ wurden in [SOP07] folgende zustandsbasierte Kriterien definiert zusammen mit Formeln für die Berechnung der Anzahl zu überdeckender Mappings:

- $Pre-states\ and\ triggers$ - Kriterium: dieses Kriterium verlangt, dass für jedes Paar von Pre-state und Trigger der aufrufenden Komponente C, also für jedes

  $(preC,\ trC)$, wobei $preC \in S_C \wedge\ trC \in Tr_C$,

  mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

$pre(t) = preC,\ tr(t) = trC.$

Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

$$\#\,Pre-states\ and\ triggers := \sum_{tr \in Tr_C} |S[Tr,E]_C(tr)|$$

- $Pre-states,\ triggers\ and\ effects$ - Kriterium: dieses Kriterium verlangt, dass für jedes Tripel von Pre-state, Trigger und Effect der aufrufenden Komponente C, also für jedes

  $(preC,\ trC,\ eC)$, wobei $preC \in S_C \wedge\ trC \in Tr_C \wedge eC \in E_C$,

  mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

  $pre(t) = preC,\ tr(t) = trC,\ e(t) = eC.$

  Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

$$\#\,Pre-states,\ triggers\ and\ effects := \sum_{\substack{tr \in Tr_C \\ e \in E_C}} |S[Tr,E]_C(tr,e)|$$

- $Invoking\ transitions$ - Kriterium: dieses Kriterium verlangt, dass für jedes Viertupel von Pre-state, Trigger, Effect und Post-state der aufrufenden Komponente C, also für jedes

  $(preC,\ trC,\ eC,\ postC)$, wobei
  $preC \in S_C \wedge\ trC \in Tr_C \wedge eC \in E_C \wedge postC \in S_C$,

  mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

  $pre(t) = preC,\ tr(t) = trC,\ e(t) = eC,\ post(t) = postC.$

  Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

$$\#\,Invoking\ transitions := |T[E]_C|$$

- $Effects\ on\ pre-states$ - Kriterium: dieses Kriterium verlangt, dass für jedes Paar von Effect der aufrufenden Komponente C und Pre-state der aufgerufenen Komponenten C', also für jedes

  $(eC,\ preC')$, wobei $eC \in E_C \wedge preC' \in S_{C'}$,

  mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

  $e(t) = eC,\ pre(t') = preC'.$

Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

$$\#\,Effects\ on\ pre-states := \sum_{e \in E_C} |S[Tr]_{C'}(e)|$$

- $Effects\ in\ /\ on\ pre-states$ - Kriterium: dieses Kriterium verlangt, dass für jedes Tripel von Pre-state und Effect der aufrufenden Komponente C und Pre-state der aufgerufenen Komponente C', also für jedes

  $(preC,\ eC,\ preC')$, wobei $preC \in S_C \wedge eC \in E_C \wedge preC' \in S_{C'}$,

  mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

  $pre(t) = preC,\ e(t) = eC,\ pre(t') = preC'$.

  Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

$$\#\,Effects\ in\ /\ on\ pre-states := \sum_{e \in E_C} |S[E]_C(e)| * |S[Tr]_{C'}(e)|$$

- $Triggers\ and\ effects\ in\ /\ on\ pre-states$ - Kriterium: dieses Kriterium verlangt, dass für jedes Viertupel von Pre-state, Trigger, Effect der aufrufenden Komponente C und Pre-state der aufgerufenen Komponente C', also für jedes

  $(preC,\ trC,\ eC,\ preC')$, wobei
  $preC \in S_C \wedge\ trC \in Tr_C \wedge eC \in E_C \wedge preC' \in S_{C'}$,

  mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

  $pre(t) = preC,\ tr(t) = trC,\ e(t) = eC,\ pre(t') = preC'$.

  Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

$$\#\,Triggers\ and\ effects\ in\ /\ on\ pre-states := \sum_{\substack{tr \in Tr_C \\ e \in E_C}} |S[Tr,E]_C(tr,e)| * |S[Tr]_{C'}(e)|$$

- $Invoking\ transitions\ on\ pre-states$ -Kriterium: dieses Kriterium verlangt, dass für jedes Fünftupel von Pre-state, Trigger, Effect und Post-state der aufrufenden Komponente C und Pre-state der aufgerufenen Komponente C', also für jedes

  $(preC,\ trC,\ eC,\ postC,\ preC')$, wobei
  $preC \in S_C \wedge\ trC \in Tr_C \wedge eC \in E_C \wedge postC \in S_C \wedge preC' \in S_{C'}$,

mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

$$pre(t) = preC,\ tr(t) = trC,\ e(t) = eC,\ post(t) = postC,\ pre(t') = preC'.$$

Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

$$\begin{aligned}\#\, Invoking\ transitions\ on\ pre-states := & \sum_{t \in T[E]_C} |S[Tr]_{C'}(e(t))| \\ = & \sum_{e \in E_C} |T[E]_C(e)| * |S[Tr]_{C'}(e)|\end{aligned}$$

- *Invoking / invoked transitions* - Kriterium: dieses Kriterium verlangt, dass für jedes Sechstupel von Pre-state, Trigger, Effect und Post-state der aufrufenden Komponente C und Pre-state und Post-state der aufgerufenen Komponente C', also für jedes

  $(preC,\ trC,\ eC,\ postC,\ preC',\ postC')$, wobei
  $preC \in S_C \wedge\ trC \in Tr_C \wedge eC \in E_C \wedge postC \in S_C \wedge preC' \in S_{C'} \wedge postC' \in S_{C'}$,

  mindestens ein Mapping $m = (t,\ t') \in Map$ überdeckt wird, für das gilt:

  $$pre(t) = preC,\ tr(t) = trC,\ e(t) = eC,\ post(t) = postC,\ pre(t') = preC',\ post(t') = postC'.$$

  Die Anzahl zu überdeckender Mappings wird anhand der folgenden Formel berechnet:

  $$\#\, Invoking\ /\ invoked\ transitions := \sum_{t \in T[E]_C} |T[Tr]_{C'}(e(t))|$$

Diese Kriterien wurden im Rahmen der Arbeit [SOP07] in eine Hierarchie eingeordnet, welche Subsumptionsrelationen enthält, die wie folgt definiert sind: ein Kriterium K subsumiert ein anderes Kriterium L, wenn jede beliebige Testfallmenge, die K erfüllt, auch L erfüllt (Notation in der Hierarchie: $K \rightarrow L$). Die aufgestellte Hierarchie wird in Abbildung 3.3 dargestellt.

Neben diesen Kriterien wurden in [SOP07] eine Reihe weiterer neuartiger Überdeckungskriterien definiert. Zum einen handelt es sich um so genannte *Mapping-basierte Kriterien*, die auf die Überdeckung einzelner Mappings oder von ganzen Mapping-Sequenzen zwischen zwei oder mehreren Komponenten zielen. Darüber hinaus werden *Message-basierte Kriterien* beschrieben, die der Tatsache Rechnung tragen, dass ein Aufruf Parameter enthalten kann. Zusätzlich zu den Mapping-basierten Kriterien, betrachten die Message-basierten Kriterien die von den Guards der aufgerufenen Transitionen auf dem Wertebereich des Parameters eingefügten Partitionen.

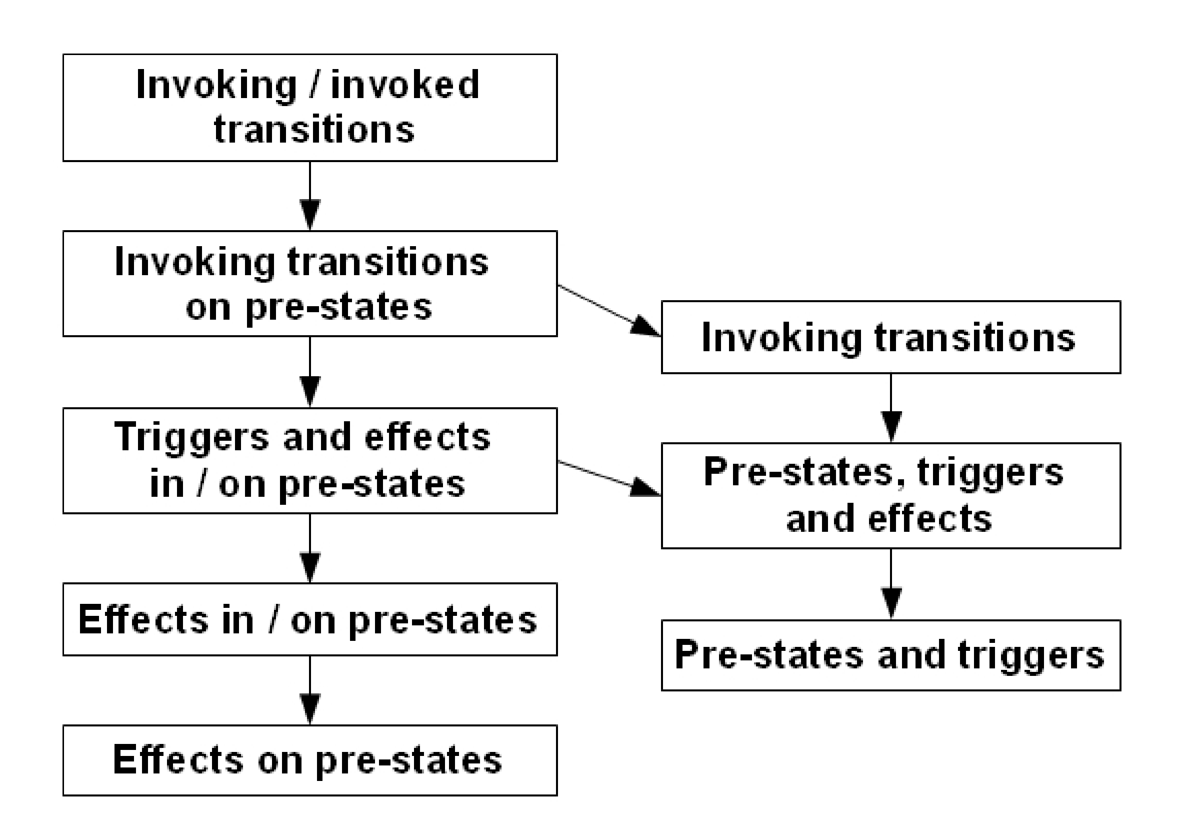


Abbildung 3.3.: Subsumptionshierarchie der zustandsbasierten Kriterien (aus [PS10])

4. Automatische, modellbasierte Testfallgenerierung mittels genetischer Algorithmen

Dieses Kapitel startet mit einem Motivationsabschnitt, in dem die allgemeine Problematik der modellbasierten Testfallgenerierung anhand von Zustandsautomaten dargestellt wird, zusammen mit der Beschreibung existierender Ansätze, die die automatische Generierung unterstützen. Dabei wird auf die Notwendigkeit des Einsatzes genetischer Algorithmen bei der Testfallgenerierung eingegangen. Danach wird in Abschnitt 4.2 die allgemeine Funktionsweise genetischer Algorithmen beschrieben, gefolgt von der Darstellung ihrer Anpassung für die Domäne der modellbasierten Testfallgenerierung in Abschnitt 4.3.

4.1. Motivation

Wie schon in Kapitel 2 erwähnt, stellt die Generierung adäquater Testfallmengen eine der aufwendigsten Aktivitäten des gesamten Test-Prozesses dar. Deshalb verspricht gerade eine weitgehende automatische Unterstützung dieser Aktivität eine verbesserte Qualität der Testfälle bei gleichzeitiger Minimierung der Kosten, die mit der Erstellung der Testfälle verbunden sind. Bei der Bewertung modellbasierter Testfallgenerierungsansätze muss differenziert betrachtet werden, welcher Teil eines Testfalls automatisch erzeugt wird: die Testsequenz und/oder die Testdaten.

Ansätze zur modellbasierten Testfallgenerierung übertragen die Idee der strukturellen Überdeckung von der Ebene des Quelltextes auf die Ebene der Modelle. Es fällt hierbei auf, dass sehr häufig die UML als Modellierungssprache gewählt wird. Dieser Eindruck wird durch die Studie von [NSVT07] bestätigt. Hier wurden 406 Artikel zum modellbasierten Test ausgewertet und 78

Ansätze ausgewählt, die genauer betrachtet wurden. Von diesen basieren 47 Ansätze, also mehr als 60%, auf UML-Modellen.

Gängige Ansätze zur modellbasierten Testfallgenerierung für den Integrationstest führen eine Transformation der unterstützten Diagramme in einen Hilfsgraphen durch, anhand dessen dann die Generierung von Testfällen durchgeführt wird. So z. B. zielt [SM10] auf die Überdeckung von Sequenzdiagrammen, indem das Diagramm in einen so genannten *Model Flow Graph* (Kurz *MFG*) transformiert wird. Dieser Hilfsgraph enthält in den Knoten die Messages aus dem Sequenzdiagramm; eine Kante im Graphen gibt die Reihenfolge wieder, in der die Messages, die die Kante verbindet, aufgerufen werden. Auch der Ansatz von [CNM07] basiert auf Transformationen des Sequenzdiagramms in einen Graphen, dem *Labeled Transition System* (Kurz *LTS*). Im Unterschied zum MFG werden die Message-Informationen an den Kanten gespeichert und nicht an den Knoten. Der in [ABuR+06] beschriebene Ansatz erlaubt die Generierung von Testsequenzen anhand des aus Zustandsautomaten und Kommunikationsdiagrammen generierten *State Collaboration Test Model* (Kurz *SCOTEM*).

Das Überdeckungsziel dieser Ansätze ist es, die Knoten bzw. die Kanten dieser Hilfsgraphen zu überdecken. Gemeinsam haben all diese Ansätze, dass sie statisch vorgehen und mittels Graphensuchalgorithmen Testsequenzen zu erzeugen erlauben.

Auch Ansätze zur Testfallgenerierung für den Komponententest, wie z. B. [KHC+99], [LI07], [WS07], [KR03] gehen statisch vor. Um Testfälle zu generieren, die Überdeckungskriterien für den Komponententest erfüllen, werden Pfade aus den Diagrammen generiert, um dann anhand der Trigger an den Transitionen dieser Pfade Testsequenzen zu erstellen.

Problematisch ist diese statische Vorgehensweise deshalb, da die Guards der Transitionen nicht berücksichtigt werden und dadurch auch Pfade entstehen, die nicht ausführbar sind. Des Weiteren müssen für die generierten Testsequenzen *geeignete Testdaten* generiert werden. Der automatischen Testdatengenerierung haben sich die Arbeiten von [LI07] und [WS07] gewidmet, indem für statisch ermittelte Testsequenzen Daten generiert werden. Der Ansatz von [WS07] stellt zunächst Constraints über die Parameter auf, um daraufhin Daten zu generieren, die diese Constraints erfüllen. Die Autoren von [LI07] setzen Heuristiken ein, um für eine gegebene Testsequenz passende Testdaten zu generieren. Für ausführbare Testsequenzen (solche, die sinnvolle Pfade im Automaten beschreiben) unterstützen diese Ansätze die Testdatengenerierung. Problematisch bei dieser Vorgehensweise ist, dass nicht ausführbare Testsequenzen im Allgemeinen nicht automatisch identifiziert werden können, was dazu führt, dass eine geeignete Datengene-

rierung unmöglich ist.

Im Gegensatz dazu, gibt der hier beschriebene Ansatz am Ende des Generierungsprozesses nur ausführbare Testsequenzen zusammen mit entsprechenden Daten aus. Weitere Alleinstellungsmerkmale dieses Ansatzes sind die Betrachtung neuartiger Integrationstestkriterien[1] und die Optimierung der Testfallmenge hinsichtlich folgender Optimierungsziele:

- *Maximierung der Überdeckung*: es soll eine Testsuite generiert werden, die eine möglichst hohe strukturelle Überdeckung des Modells erreicht (hinsichtlich klassischer Überdeckungskriterien für den Komponententest und den in Unterabschnitt 3.3.2 beschriebenen zustandsbasierten Überdeckungskriterien);
- *Minimierung der Testfallanzahl*: dies ist deshalb besonders interessant, da trotz automatischer Testfallgenerierung die Bewertung der Testfälle (Zuordnung eines *Verdicts*) als bestanden oder nicht bestanden noch sehr oft manuell durch den Tester erfolgen muss. Hier entsteht also auch ein nicht unerheblicher Aufwand, der durch die Minimierung der Anzahl auszuführender Testfälle reduziert werden kann.

Bei der Testfallgenerierung handelt es sich also um ein *Optimierungsproblem*, bei dem mehrere Optimierungsziele gleichzeitig verfolgt werden. Erschwerend kommt hinzu, dass es sich um zwei *rivalisierende Ziele* handelt: eine Testfallmenge mit vielen Testfällen kann im Allgemeinen leichter eine hohe Überdeckung erreichen, als eine mit wenigen Testfällen. Diese zwei Optimierungsziele sind aber nicht gleich wichtig: eine hohe Überdeckung hat wesentlich höhere Priorität als die Minimierung der Testfallanzahl, da durch eine hohe Überdeckung die Wahrscheinlichkeit, Fehler zu finden, erhöht wird. Besonders im Kontext sicherheitskritischer Anwendungen wird die Anzahl auszuführender Testfälle sekundär sein, falls durch mehr Testfälle die Fehlererkennungswahrscheinlichkeit erhöht wird.

Zur Lösung solcher Optimierungsprobleme eignen sich unter anderem *genetische Algorithmen*, welche als so genannte *Meta-Heuristiken* heuristische Verfahren sind, die der Suche nach guten (nahezu optimalen) Lösungen für verschiedene Optimierungsprobleme in möglichst kurzer Zeit dienen. Meta-Heuristiken werden sie deshalb genannt, da sie Lösungsansätze für Optimierungsprobleme im Allgemeinen beschreiben, was dazu führt, dass sie für den Einsatz im konkreten Anwendungsgebiet angepasst werden müssen.

Der hier beschriebene Ansatz mit genetischen Algorithmen basiert auf am Lehrstuhl für Soft-

[1] Siehe Abschnitt 3.3

ware Engineering vorausgegangene Arbeiten: der Ansatz von [Ost07] erlaubt die vollautomatische Testfallerstellung für Java-Code und unterstützt kontroll- und datenflussbasierte Code-Überdeckungskriterien[2]. Im Rahmen der Arbeit [Ost07] werden ebenfalls die Ziele der Überdeckungsmaximierung bei gleichzeitiger Minimierung der benötigten Anzahl an Testfällen verfolgt. Einerseits werden diese beiden Ziele völlig unabhängig voneinander betrachtet, indem die so genannte *Pareto-Front* im Rahmen der Verfahrens *Nondominated Sorting Genetic Algorithm* (Kurz *NSGA*) optimiert wird. Diese Front beschreibt mehrere pareto-optimale Testsuites, wobei eine Testsuite dann pareto-optimal ist, wenn es keine andere Testsuite gibt, die diese Testsuite hinsichtlich beider Eigenschaften dominiert[3]. Am Ende des Generierungsprozesses stehen dem Tester also eine Menge pareto-optimaler Testsuites zur Verfügung. Aus dieser Menge kann der Tester dann die Testsuite auswählen, die für ihn den besten Kompromiss zwischen Überdeckung und Testaufwand bietet.

Andererseits werden Verfahren vorgestellt (*Random Search*, *Simulated Annealing*, *Multi-objective Aggregation*), die diese zwei Ziele zu einer einfachen Zielfunktion durch eine gewichtete Summe zusammenfassen. Am Ende der Generierung steht in diesem Fall eine einzige Testsuite zur Verfügung. Diese Vorgehensweise nennt der Autor *pseudo-multi-objektiv*, da die Ziele nicht völlig unabhängig voneinander (wie im Falle des Verfahrens *NSGA*) betrachtet werden. Bei dieser Vorgehensweise müssen – im Gegensatz zum NSGA-Verfahren – vor dem Start der Generierung die Gewichte für die Zielfunktion festgelegt werden.

Nach der Definition aus [Ost07] ist das in Abschnitt 4.3 beschriebene Verfahren auch als *pseudo-multi-objektiv* zu bezeichnen. Im Folgenden beschreibt Abschnitt 4.2 die allgemeine Vorgehensweise genetischer Algorithmen, gefolgt von der Beschreibung der Anpassung der Algorithmen an die Domäne der modellbasierten Testfallgenerierung.

4.2. Allgemeine Funktionsweise genetischer Algorithmen

Genetische Algorithmen imitieren evolutionäre Prozesse, die in der Natur zu beobachten sind und von Charles Darwin in seiner Evolutionstheorie [Dar59] beschrieben wurden. Diese Algorithmen wurden von John Holland in [Hol75] eingeführt und stützen sich auf das Prinzip: *Survival of*

[2] Siehe Abschnitt 2.5.2

[3] D. h., dass es keine andere Testsuite gibt, die sowohl eine höhere Überdeckung erreicht als auch weniger Testfälle enthält

the fittest. Dies bedeutet, dass im Laufe der Evolution nur die Individuen überleben, die an die äußeren Bedingungen am besten angepasst sind (also eine hohe *Fitness* haben). Somit können hauptsächlich solche gut angepassten Individuen ihre Erbmerkmale weitergeben, was dazu führt, dass die nächste Generation üblicherweise besser an die äußeren Bedingungen angepasst sein wird, als die vorangegangene.

Der größte Vorteil genetischer Algorithmen besteht darin, dass sie universell einsetzbar sind, d. h. auch für Probleme, die systematisch nicht gelöst werden können. Ein Nachteil dieser Algorithmen ist, dass sie nicht garantieren, eine optimale Lösung zu finden. Sie erlauben es jedoch innerhalb eines akzeptablen Zeitraums optimierte Lösungen zu finden.

Die Algorithmen funktionieren nach folgendem generischem Prinzip: Zunächst wird eine *Anfangspopulation* von Individuen zufällig generiert, die anschließend im Hinblick auf ihre Fitness bewertet wird. Aus dieser *Anfangspopulation* geht durch sukzessive Anwendung genetischer Operatoren (*Elitismus*, *Selektion*, *Rekombination* und *Mutation*) eine neue Population hervor, die die Anfangspopulation überschreibt. Diese neu generierte Population wird hinsichtlich ihrer Fitness bewertet, wonach eine neue Population durch erneute Anwendung der genetischen Operatoren erzeugt wird. Dieser Vorgang wird iterativ fortgesetzt und terminiert, sobald einer der generierten Individuen einer Population einen vorgegebenen Fitnesswert erzielt hat bzw. ein vorgegebenes Abbruchkriterium (etwa eine vorgegebene Anzahl an Iterationen) erreicht wurde. Skizziert wird dieses Vorgehen in Abbildung 4.1.

Zentral für die Effizienz solcher Algorithmen sind also die Operatoren, die aus einer bestehenden Population eine neue Population erzeugen sollen. Dazu werden die Operatoren so lange angewendet, bis die neue Population eine vorgeschriebene Populationsgröße erreicht. Im Folgenden werden die genetischen Operatoren allgemein beschrieben, bevor die konkrete Implementierung der Operatoren für das im Rahmen dieser Arbeit betrachtete Problem in Abschnitt 4.3 beschrieben wird.

Elitismus

Grundidee der genetischen Algorithmen ist es, aus einer bestehenden Population durch Anwendung genetischer Operatoren auf *Eltern-Individuen* neue *Kind-Individuen* zu generieren, die die neue Population bilden. Bei der Erzeugung neuer Individuen kann es vorkommen, dass diese eine schlechtere Fitness als ihre Eltern-Individuen haben. Dies kann dazu führen, dass die neu erzeug-

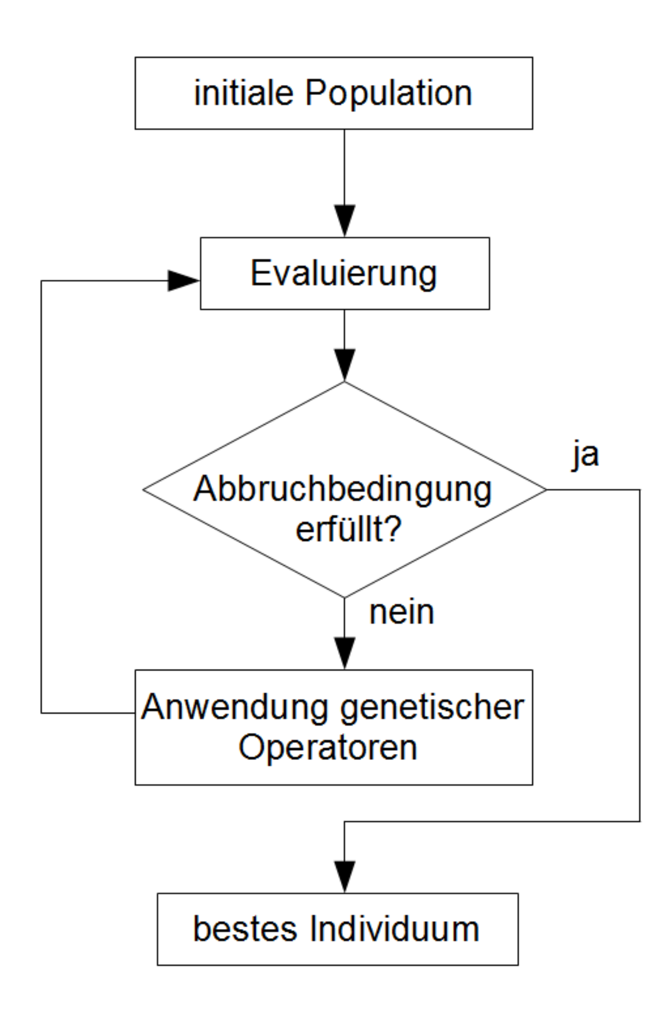


Abbildung 4.1.: Allgemeine Vorgehensweise genetischer Algorithmen (aus [SP10])

te Population schlechter ist als die alte Population[4]. Um dies zu vermeiden, ist die unveränderte Übernahme eines Teils der fittesten Individuen der alten Population in die neue Population sinnvoll. Dies wird durch den so genannten *Elitismusoperator* realisiert, der als erster angewendet wird und einen Teil der besten Individuen der alten Population in die neue Population unverändert kopiert.

Selektion

Die Eltern-Individuen werden mit Hilfe der *Selektionsoperatoren* ausgesucht. Grundsätzlich wählen solche Operatoren aus der aktuellen Generation die besten Individuen aus, also diejenigen mit der höchsten Fitness. Um aber zu vermeiden, dass sich die Optimierung hin zu einem *lokalen* Maximum bewegt, sollten Selektionsoperatoren auch Individuen auswählen, die eine schlechte Fitness haben. Aufgrund der unterschiedlichen Eigenschaften von schlechten und guten Individuen kann es dann vorkommen, dass durch ihre Kombination ein sehr gutes Individuum entsteht. Es gibt mehrere Selektionsstrategien, wobei ein paar hier kurz erwähnt werden.

Die *Best Fitness* Strategie besagt, dass für die Erzeugung neuer Individuen, immer die besten zwei Individuen der aktuellen Population herangezogen werden sollen. Dies ist aufgrund der vorhin beschriebenen Betrachtung zu der Kombination von schlechten und guten Individuen nicht sinnvoll, da etwas schlechtere Individuen durch diese Strategie gar nicht beachtet werden.

Um auch schwächere Individuen bei der Auswahl zu berücksichtigen, beruht die *Roulette Wheel*[5] Strategie auf die Zuweisung von Auswahlwahrscheinlichkeiten zu einzelnen Individuen, wobei die Wahrscheinlichkeit pro Individuum proportional zu der Fitness des Individuums innerhalb der aktuellen Population ist.

Tournament Selection ist ein weiteres Beispiel für eine Strategie, die auch schwächere Individuen berücksichtigt. Um ein Individuum auszuwählen, werden im Rahmen dieser Strategie zunächst zufällig Individuen aus der aktuellen Generation ausgewählt. Aus der Menge der ausgewählten Individuen wird das Individuum mit der höchsten Fitness ermittelt. Daher auch der Name, der suggeriert, dass mehrere Auswahlvorgänge (Turniere) ausgeführt werden, wobei am Ende jedes Turniers ein Sieger (das fitteste Individuum) selektiert wird.

---

[4] Dies bedeutet, dass das fitteste Individuum der alten Population eine höhere Fitness hat, als das fitteste Individuum der neuen Population

[5] Wird auch *Fitness proportionate selection* genannt

Rekombination

Um ausgehend von den Eltern-Individuen neue Kind-Individuen zu erzeugen, werden *Rekombinationsoperatoren* eingesetzt. Bezüglich der Reihenfolge in der Selektion und Rekombination ausgeführt werden, gibt es zwei Möglichkeiten. Die eine Möglichkeit ist, immer nur zwei Eltern-Individuen zu selektieren, aus denen dann durch Rekombination ein oder zwei neue Kind-Individuen erzeugt werden. Alternativ kann der Selektionsoperator eine ganze Menge von potentiellen Eltern generieren, wobei der Rekombinationsoperator daraufhin in jedem Durchlauf zwei Individuen aus dieser Menge auswählt.

Zur Erzeugung der Kind-Individuen gibt es mehrere Vorgehensweisen. Die einfachste Variante ist der *1-point-crossover*-Operator, der die zwei Eltern zunächst in die neue Population kopiert, dann einen beliebigen Kreuzungspunkt auswählt, um dann die Bestandteile der Individuen ab diesem Punkt auszutauschen. Ähnlich geht der *2-point-crossover*-Operator vor, indem er zwei Kreuzungspunkte setzt. Die Menge an Kreuzungspunkten kann auch beliebig erweitert werden, so dass im allgemeinen Fall von einem *n-point-crossover* gesprochen wird.

Mutation

Zur Erzeugung einer neuen Population wird der Rekombinationsoperator in Kombination mit dem Selektionsoperator mehrmals angewendet. Würde sich der Algorithmus nur auf diese zwei Operatoren beschränken, wären die neu erzeugten Individuen nur durch Kombinationen der Individuen entstanden, die in der zufällig generierten Anfangspopulation vorhanden waren. D. h., die zufällig generierte Anfangspopulation wäre ausschlaggebend für den Erfolg der Suche.

Um dies zu vermeiden, wird ein zusätzlicher genetischer Operator benötigt, und zwar der *Mutationsoperator*. Dieser fügt in die neu generierten Individuen kleine Änderungen ein, so dass Individuen entstehen können, die nicht als Kombination der Individuen aus der Anfangspopulation beschrieben werden können.

4.3. Einsatz genetischer Algorithmen zur automatischen, modellbasierten Testfallgenerierung

Als Meta-Heuristiken beschreiben genetische Algorithmen ein allgemeines Verfahren zur Suche nach Lösungen für Optimierungsprobleme. Um diese Algorithmen zur Lösung eines speziellen

Problems einsetzen zu können, müssen sie an das Problemgebiet angepasst werden. D. h., es muss festgelegt werden wie:

- die *Individuen* kodiert werden
- die genetischen Operatoren (*Elitismus*, *Selektion*, *Rekombination*, *Mutation*) vorgehen, d. h. es muss festgelegt werden, wie die Individuen durch Anwendung dieser Operatoren ausgewählt bzw. verändert werden
- die *Anfangspopulation* generiert wird
- die *Fitnessfunktion* definiert ist
- die *Abbruchbedingung* des Algorithmus lautet.

Im Folgenden wird auf die Anpassung der genetischen Algorithmen für die modellbasierte Testfallgenerierung unter Berücksichtigung dieser Punkte einzeln eingegangen.

4.3.1. Kodierungsvorschrift für Individuen

Eingesetzt im Kontext der automatischen Testfallgenerierung zielt der genetische Algorithmus auf die Generierung einer möglichst optimalen Testsuite. Durch Übertragung der Terminologie der genetischen Algorithmen auf die Terminologie der automatischen Testfallgenerierung, ergibt sich folgende Zuordnung, die in Abbildung 4.2 bildlich dargestellt wird:

Gen = Testfall

Individuum = Testsuite

Population = Menge von Testsuites

Wie schon in Abschnitt 2.5 erwähnt, ist ein Testfall eine Folge von Aufrufen an die zu testende Komponente(n). Ein Komponententestfall beinhaltet Aufrufe an eine einzige Komponente. Ein Beispiel für einen Komponententestfall für Komponente *A* aus Abbildung 3.2 ist:

opA1(88742965), opA4(true), opA5()[6]

Dagegen beinhaltet ein Integrationstestfall Aufrufe an alle interagierenden Komponenten. Dabei sind die *internen Aufrufe* zwischen den Komponenten nicht im Integrationstestfall mit enthalten. So z. B. kann ein Integrationstestfall für die Komponenten *A* und *B* aus Abbildung 3.2 die

[6]Die Überdeckungsbestimmung dieses Komponententestfalls wird in Abschnitt 4.3.3.1 beschrieben

```
opA1(-1104924320), opB4(), opA5()
opA1(1456540800), opB4(), opA2(), opA3(), opA2(), opA3(), opA4(true), opA5()
opA1(-129423737), opB4(), opA2(), opA3(), opA4(true), opA5()
opA1(-1787143425), opB4(), opA5()
opA5(), opB4()

opA1(1500943729), opB1(-2050836360), opA5(), opB4()
opA5(), opB4()
opA5(), opB1(-1555817572), opB4()
opA5(), opB1(1329393341), opB4()
.
.
.
opA1(-1924151534), opB1(2076310642), opA2(), opA3(), opB4(), opA2(), opA3(), opA2(), opA3(), opA2(), opA3()
opA1(1770596331), opB1(1627284803), opA5(), opB4()
opA1(-1988063922), opB4(), opA2(), opA3(), opA2(), opA3(), opA2(), opA3(), opA2(), opA3()
opA1(1794321776), opB1(1052368042), opA5(), opB4()
opA5(), opB1(-620774538), opB4()
```


Abbildung 4.2.: Kodierungsvorschrift der Individuen

Aufrufe $opB1(12345)$, $opB3()$ und $opB4()$ an Komponente B enthalten, aber nicht den Aufruf $opB2()$, da $opB2()$ ein interner Aufruf zwischen den Komponenten A und B ist. Ein Beispiel für einen Integrationstestfall für die Komponenten A und B ist:

opA1(4), opB1(69171531), opA2(), opA3()[7]

Im Rahmen dieser Arbeit wird als *globale Optimierung* das Vorgehen bezeichnet, das den in Abbildung 4.1 beschriebenen Algorithmus auf Mengen von Testsuites umsetzt.

Während der globalen Optimierungsphase werden also die zwei Ziele (Maximierung der Überdeckung und Minimierung der Testfallanzahl) gleichzeitig verfolgt. Dabei kann es vorkommen, dass manche Entitäten der Zustandsautomaten während der globalen Optimierungsphase nicht überdeckt werden. Deshalb wurde auch eine *lokale Optimierung* umgesetzt, welche dazu dient, Transitionen oder Zustände eines Zustandsautomaten zu überdecken, die während der globalen Optimierungsphase nicht überdeckt wurden. Die Suche nach nicht überdeckten Mappings wird allerdings nicht unterstützt. In Rahmen der lokalen Optimierungsphase wird also nur das Ziel der Überdeckungsmaximierung verfolgt.

Wann die lokale Optimierung eingesetzt wird, wird über den *Stagnationsparameter* gesteuert. Dieser muss vor dem Start der Generierung festgelegt werden und gibt an, nach wie vielen Generationen ohne Überdeckungsfortschritt der globalen Optimierung die lokale Optimierung eingeschaltet werden soll. Die Abbruchbedingung für die lokale Optimierung wird über zwei weitere Parameter gesteuert. Der lokale Stagnationsparameter gibt an, nach wie vielen Generationen ohne Fitnessfortschritt der lokalen Optimierung zurück zur globalen Optimierung gewechselt wird.

[7] Die Überdeckungsbestimmung dieses Integrationstestfalls wird in Abschnitt 4.3.3.1 beschrieben

Ein weiterer Parameter gibt die Anzahl der lokalen Generationen an, nach denen auf jeden Fall zurück zur globalen Optimierung gewechselt wird.

Während der lokalen Optimierungsphase werden einzelne Testfälle als Individuen angesehen; der umgesetzte Basisalgorithmus ist aber derselbe, jedoch mit einer unterschiedlichen Fitnessfunktion und einer Einschränkung in der Wahl der genetischen Operatoren. Von den verfügbaren Rekombinations- und Mutationsoperatoren werden im lokalen Optimierungsschritt nur die Operatoren betrachtet, die auf *Aufruf*- und *Testdatenebene*[8] angewendet werden können. Der Elitismus- und Selektionsoperator funktionieren ähnlich wie bei der globalen Optimierung. Die Unterschiede in der Art der Fitnessbewertung einer Testsuite und eines Testfalls werden in Unterabschnitt 4.3.3 beschrieben.

4.3.2. Anfangspopulation

Der genetische Algorithmus startet mit der Generierung einer Anfangspopulation, indem eine vor dem Start der Generierung festgelegte Anzahl an Individuen generiert wird. Jedes einzelne Individuum wird wiederum generiert, indem eine vorgegebene Anzahl an Testfällen erzeugt wird, wobei jeder Testfall eine vor dem Start der Generierung festgelegte Anzahl an Aufrufen nicht überschreiten darf. Die Generierung einer Anfangspopulation von Komponententestfällen unterscheidet sich von der Generierung einer Anfangspopulation von Integrationstestfällen.

Testfälle für den Komponententest werden generiert, indem als erstes *Testsequenzen* erstellt werden. Dazu wird erst statisch eine Folge von Transitionen durch den Zustandsautomat ermittelt. Der Algorithmus geht dabei schrittweise wie folgt vor: in jedem Schritt wird ein Zustand[9] des Automaten betrachtet und aus der Menge der daraus ausgehenden Transitionen eine Transition zufällig ausgewählt. Eine Nachricht, die gleich dem Trigger der ausgewählten Transition ist, wird der Testsequenz hinzugefügt[10]. Danach wird der Post-state der gerade ausgewählten Transition betrachtet und der Vorgang wiederholt, indem wieder zufällig eine Transition ausgewählt wird. Dies passiert so oft, bis für jede Testsequenz genügend Nachrichten erzeugt wurden oder ein Endzustand erreicht wird, für den keine ausgehenden Kanten existieren.

Zum Beispiel könnte der Algorithmus anhand des Zustandsautomaten der Komponente *A* aus

[8] Siehe Unterabschnitt 4.3.4

[9] Der erste betrachtete Zustand ist der Anfangszustand des Zustandsautomaten

[10] Falls die ausgewählte Transition keinen Trigger hat, wird keine Nachricht hinzugefügt

Abbildung 3.1 folgende Transitionen nacheinander auswählen: *tA1, tA3, tA7, tA8*. Dem entsprechend ist ein Beispiel für eine *Testsequenz*, die anhand dieses Zustandsautomaten generiert werden könnte und 3 Nachrichten enthält, folgende:

opA1, opA4, opA5.

Für jede auf diese Art generierte *Testsequenz* werden zu jeder enthaltenen Nachricht zufällig *Testdaten* generiert. Da *opA1* und *opA4* parametrisiert sind, wird für *opA1* ein zufälliger Integerwert und für *opA4* ein zufälliger boolescher Wert generiert. Somit könnte ein solcher *Testfall* wie folgt aussehen:

opA1(-1230053765), opA4(false), opA5().

Bei der Generierung einer Anfangspopulation von Integrationstestfällen werden für alle interagierenden Zustandsautomaten wie oben beschrieben einzeln Testfälle generiert, mit einem Unterschied: falls eine Transition t ausgewählt wird, die von Transitionen aus anderen Zustandsautomaten aufgerufen wird (falls ein Mapping $m = (t_1, t) \in Map$ existiert), wird der Trigger der Transition t der Testsequenz *nicht* hinzugefügt. Dies ist dadurch bedingt, dass diese Transition als Folge interner Komponentenaufrufe durchlaufen wird und nicht durch den Testfall aufgerufen werden kann. Als Beispiel für eine Folge von Transitionen, die aus dem Zustandsautomaten der Komponente B (aus Abbildung 3.2) generiert wurde, sei: *tB1, tB2, tB3, tB7*. Der dadurch generierte Testfall ist[11]:

opB1(69171531), opB4().

Die generierten Testfälle aus A und B rufen also nur jeweils eine Komponente auf. Um daraus einen Integrationstestfall zu generieren, werden diese zwei Testfälle zu einem einzigen Testfall zusammengefasst, indem nacheinander Aufrufe aus dem ersten und aus dem zweiten Testfall dem neu erzeugten Integrationstestfall hinzugefügt werden. Ein Beispiel für einen Integrationstestfall ist:

opA1(-1230053765), opB1(69171531), opA4(false), opB4(), opA5().

Durch die statische Vorgehensweise bei der Generierung der Anfangspopulation werden auch Testfälle generiert, die nicht komplett abgearbeitet werden können. D. h., dass manche Aufrufe des Testfalls bei der Ausführung auf dem entsprechenden Zustandsautomaten verfallen, da keine passende Transition aus dem aktuellen Zustand ausgeht. Der gerade beschriebene Testfall ist

[11] Der Trigger *opB2()* von Transition *tB3* wurde nicht hinzugefügt, da dieser als Effect der Transition *tA6* aufgerufen wird

ein Beispiel dafür. Der Aufruf *opA4(false)* kann nicht bearbeitet werden, da die Komponente *A* nach dem Aufruf von *opA1(-1230053765)* (mit einem Wert kleiner als 0) in den Zustand *A3* übergeht, in dem der Aufruf *opA4(false)* nicht bearbeitet werden kann. Durch Modellsimulation und die darauf folgende Anwendung genetischer Operatoren werden allerdings ausgehend von der Anfangspopulation Testfälle erzeugt, die komplett abgearbeitet werden können und eine hohe Überdeckung erreichen.

4.3.3. Fitnessfunktion der Individuen

Ein wichtiges Konzept der genetischen Algorithmen ist die verwendete *Fitnessfunktion*. Sie steuert den Algorithmus maßgeblich, indem sie jedem generierten Individuum einen numerischen Wert zuordnet, der die Güte dieses Individuums hinsichtlich der angestrebten Optimierungsziele angibt. Somit wird ermöglicht, dass einzelne Individuen einer Population miteinander verglichen werden.

Wie schon in Abschnitt 4.3.1 erwähnt, wird zwischen globaler und lokaler Optimierung unterschieden. Dementsprechend gibt es auch eine globale und eine lokale Fitnessfunktion. Da die lokale Optimierung nur eine Ergänzung zur globalen Optimierung darstellt und darüber hinaus nur den Komponententest unterstützt, ist sie für den Zweck dieser Arbeit nicht von so hoher Bedeutung. Deshalb wird im Folgenden die globale Fitnessfunktion detailliert dargestellt, wohingegen die Beschreibung der lokalen Fitnessfunktion bewusst kompakt gehalten wird.

Die *globale Fitnessfunktion* berechnet einen Fitnesswert, der als *gewichtete Summe* über die folgenden Eigenschaften der *Testsuite* berechnet wird:

- *Anteil an überdeckten Entitäten*
- *Anzahl erforderlicher Testfälle in Relation zur Population.*

Zur Bewertung einer Testsuite benötigt die Fitnessfunktion also Informationen zu der erreichten Modellüberdeckung. Dazu werden zum einen durch statische Analyse des Modells, die zu überdeckenden Modellentitäten bestimmt. Zur Ermittlung der *erreichten Testüberdeckung* muss danach bestimmt werden, welche der angestrebten Entitäten von der betrachteten Testsuite überdeckt wurden. Dieses wird durch Modellsimulation realisiert[12]. Anschließend wird die erreichte *Modellüberdeckung* als Verhältnis zwischen der Anzahl der von der Testsuite TS_i überdeckter

[12] Siehe 4.3.3.1

Entitäten ($A\ddot{U}(TS_i)$) durch die Anzahl der zu überdeckenden Entitäten ($AZ\ddot{U}$) berechnet:

$$E\ddot{U}(TS_i) = \frac{A\ddot{U}(TS_i)}{AZ\ddot{U}} \tag{4.1}$$

Der Wert für die Anzahl der Testfälle wird durch *Windowing* in Relation zu den anderen Testsuites der Population wie folgt berechnet:

$$AT(TS_i) = \frac{Q_{max} - Q_i}{Q_{max} - Q_{min}} \tag{4.2}$$

Dabei ist Q_{min} die Anzahl der Testfälle des Individuums, das die wenigsten Testfälle der ganzen Population enthält und Q_{max} die Anzahl der Testfälle des Individuums, das die meisten Testfälle der ganzen Population enthält. Q_i ist die Anzahl der Testfälle des betrachteten Individuums, also der Testsuite TS_i.

Basierend auf den gerade beschriebenen Konzepten, wird nun die Formel für die Berechnung der Fitness einer Testsuite TS_i angegeben:

$$F(TS_i) = G\ddot{U} \cdot E\ddot{U}(TS_i) + GT \cdot AT(TS_i),\ wobei\ G\ddot{U}, GT \in [0,1] \wedge G\ddot{U} + GT = 1 \tag{4.3}$$

Dabei steht $G\ddot{U}$ für das Gewicht, das für die Überdeckung festgelegt wurde, $E\ddot{U}(TS_i)$ steht für die erreichte Überdeckung; dieser Wert wird anhand der Formel 4.1 berechnet. GT steht für die Gewichtung, die für die Testfallanzahl festgelegt wurde und $AT(TS_i)$ für einen Wert, der anhand der in der Testsuite enthaltenen Anzahl an Testfällen nach der Formel 4.2 berechnet wird. Die Gewichte $G\ddot{U}$ und GT müssen vor dem Start der Testfallgenerierung festgelegt werden, wobei die einzelnen Gewichte den Wertebereich $[0,1]$ haben und die Summe der Gewichte 1 ist.

Die *lokale Fitnessfunktion* bewertet jeden Testfall, indem der durch den Testfall überdeckte Pfad betrachtet wird. Unter anderem wird berechnet wie „weit" sich das letzte Element dieses Pfades vom gesuchten Element befindet. Dazu wird statisch ermittelt, wie viele Kanten mindestens vom letzten Element, das durch den Testfall überdeckt wird, bis zum Element, das überdeckt werden soll, durchlaufen werden müssen. Danach wird berechnet, wie viele Aufrufe im Testfall fehlen, um diese Kanten zu überdecken. Des Weiteren wird berechnet, inwiefern die Guards an diesen Kanten durch den aktuellen Testfall erfüllt werden. Dadurch, dass diese Werte einzelnen Testfällen zugeordnet werden, können einzelne Testfälle miteinander verglichen werden.

4.3.3.1. Überdeckungsbestimmung durch Simulation

Für die Berechnung der Fitness einer Testsuite, muss unter anderem die von der Testsuite erreichte Überdeckung bestimmt werden. Dieses wird realisiert, indem für jeden darin enthaltenen Testfall die Überdeckung gemessen wird. Dafür wurde ein *Modellsimulator* im Rahmen der Arbeit [Sch06] umgesetzt, der Zustandsautomaten zu simulieren erlaubt[13], in der Sprache Java implementiert wurde und als Komponente in das Werkzeug UnITeD[14] integriert ist.

Für die Bestimmung der überdeckten Entitäten eines Komponententestfalls wird ein einziger Simulator für die Komponente instanziiert. Im Gegensatz dazu, wird bei der Bestimmung der überdeckten Entitäten von Integrationstestfällen für jede interagierende Komponente ein Simulator instanziiert, der dann die Aufrufe aus dem Testfall verarbeitet, die zu der Komponente gehören, die er simuliert.

Die Verwaltung der Simulatorinstanzen übernimmt ein so genannter *SimulationManager*. Die Testsuite, für die die Überdeckung bestimmt werden soll, wird an diesen SimulationManager geschickt. Der SimulationManager verarbeitet jeden einzelnen in dieser Testsuite enthaltenen Testfall, indem er die enthaltenen Aufrufe an den entsprechenden Simulator schickt. Danach werden pro Testfall die überdeckten Entitäten von den Simulatoren abgefragt und gespeichert. Durch die Vereinigung der Menge aller Entitäten, die durch mindestens einen Testfall überdeckt werden, wird die Menge der Entitäten aufgebaut, die von der Testsuite überdeckt werden.

Beispielhaft wird die *Überdeckungsbestimmung eines Komponententestfalls* für die Komponente *A* anhand des folgenden Testfalls beschrieben:

opA1(88742965), opA4(true), opA5().

Die Schritte, die vom SimulationManager zur Bestimmung der Überdeckung dieses Komponententestfalls durchgeführt werden, sind:

- Schritt 1: Instanziierung und Initialisierung des Simulators von *A*. Komponente *A* ist im Zustand *A1*, da die Transition *tA1* keinen Trigger besitzt und somit gleich durchlaufen wird.
- Schritt 2: Bearbeitung des ersten Aufrufs der Testfalls: *opA1(88742965)*. Dieser Aufruf

[13] Dieser Simulator unterstützt auch Aktivitätsdiagramme; diese Diagrammart wurde allerdings in den Modellen, die im Rahmen dieser Arbeit betrachtet werden, nicht benutzt

[14] Siehe Kapitel 5

wird an den Simulator von *A* geschickt. Der Guard von *tA3* wird zu *true* ausgewertet (weil 88742965 größer als 0 ist) was dazu führt, dass die Transition *tA3* traversiert wird, wobei der internen Variablen *x* der Wert 88742965 zugewiesen wird. Der neue aktive Zustand von *A* ist *A2*.

- Schritt 3: Bearbeitung des zweiten Aufrufs des Testfalls: *opA4(true)*. Dieser wird an den Simulator von *A* geschickt; da aus dem aktiven Zustand *A2* die Transition *tA7* mit Trigger *opA4(j:boolean)* existiert und der Guard der Transition erfüllt ist, wird diese Transition als Folge dieses Aufrufs durchlaufen. Somit ist Zustand *A3* der neue aktuelle Zustand der Komponente *A*.
- Schritt 4: Bearbeitung des dritten Aufrufs des Testfalls: *opA5()*. Dieser wird an den Simulator von *A* geschickt; da aus dem aktiven Zustand *A3* die Transition *tA8* mit Trigger *opA5()* existiert, wird diese Transition als Folge dieses Aufrufs durchlaufen. Somit ist Zustand *A5* der neue aktuelle Zustand der Komponente *A*.

Dieser Testfall überdeckt also folgende Transitionen: *tA1, tA3, tA7, tA8*. D. h., dass von den insgesamt acht Transitionen durch diesen Testfall vier Transitionen überdeckt werden. Eine Testsuite, die nur diesen Testfall enthält, erreicht also eine 50%ige Transitionsüberdeckung. Um alle Transitionen zu überdecken, müsste die Testsuite um zusätzliche Testfälle erweitert werden; z. B. um folgende drei Testfälle:

opA1(4), opA2(), opA3() (überdeckt zusätzlich die Transitionen *tA5* und *tA6*)
opA1(-1230053765) (überdeckt zusätzlich die Transition *tA4*)
opA5() (überdeckt zusätzlich die Transition *tA2*).

Am folgenden Beispiel wird gezeigt, wie die Überdeckung eines Integrationstestfalls für die Komponenten *A* und *B* aus Abbildung 3.2 gemessen wird. Überdeckungsziel ist Überdeckung der zwei Mappings *m1* und *m2*. Der betrachtete Testfall sei:

opA1(4), opB1(69171531), opA2(), opA3().

Die Bestimmung der überdeckten Entitäten wird im Folgenden Schritt für Schritt erklärt:

- Schritt 1: Instanziiere und initialisiere die Simulatoren. Komponente *A* ist im Zustand *A1* und Komponente *B* ist in Zustand *B1*, da die Transitionen *tA1* und *tB1* keinen Trigger besitzen und somit gleich durchlaufen werden.
- Schritt 2: Bearbeitung des ersten Aufrufs der Testfalls: *opA1(4)*. Dieser Aufruf wird an den

Simulator der Komponente *A* geschickt. Der Guard von *tA3* wird zu *true* ausgewertet (weil 4>0 ist), der internen Variablen *x* wird der Wert 4 zugewiesen und die Transition *tA3* wird traversiert. Der neue aktive Zustand von *A* ist *A2*.

- Schritt 3: Bearbeitung des zweiten Aufrufs des Testfalls: *opB1(69171531)*. Dieser wird an den Simulator von *B* geschickt. Der internen Variablen *y* von *B* wird der Wert 69171531 zugewiesen, die Transition *tB2* wird traversiert. Somit ist *B2* der neue aktive Zustand von *B*.
- Schritt 4: Bearbeitung des dritten Aufrufs des Testfalls: *opA2()*. Dieser wird an den Simulator der Komponente *A* verschickt. Da der Guard der Transition *tA5* zu *true* ausgewertet (weil $x = 4$, was kleiner als 5 ist) wird, wird diese Transition durchlaufen, was den neuen aktiven Zustand *A4* der Komponente *A* zur Folge hat.
- Schritt 5: Bearbeitung des vierten Aufrufs des Testfalls: *opA3()*. Dieser wird an den Simulator der Komponente *A* geschickt. Transition *tA6* wird durchlaufen, was zu dem neuen aktiven Zustand *A2* führt. Der Effect dieser Transition enthält einen Aufruf der Komponente *B*. Der Guard der Transition *tB4* wird zu *true* evaluiert (weil $y = 69171531$ und somit größer als 0 ist), was dazu führt, dass *tB4* traversiert wird. Daraus resultiert der neue aktive Zustand *B3* von Komponente *B*.

Die von diesem Testfall überdeckten Transitionen sind:

tA1, tB1, tA3, tB2, tA5, tA6, tB4.

Anhand dieser Liste an überdeckten Transitionen lässt sich feststellen, dass die Transition *tB4* als Folge des Durchlaufens der Transition *tA6* traversiert wurde. Dieser Testfall hat also das Mapping *m1 = (tA6, tB4)* überdeckt. Da es insgesamt zwei Mappings gibt, beträgt die von diesem Testfall erreichte Überdeckung (bezüglich dem Kriterium *Invoking / invoked transitions*) 50%. Zur Überdeckung des anderen Mappings müsste ein zusätzlicher Testfall generiert werden, z. B.:

opA1(4), opA2(), opB1(-1774418), opA3()

Dieser Testfall überdeckt folgende Transitionen:

tA1, tB1, tA3, tA5, tB2, tA6, tB3.

Um die gewünschte Überdeckung mit einem statt zwei Testfällen zu erreichen, müsste der erste Testfall so adaptiert werden, dass er beide Mappings überdeckt. Ein Beispiel für einen Testfall, der beide Mappings überdeckt ist:

opA1(4), opB1(69171531), opA2(), opA3(), opB3(), opB1(-1774418), opA2(), opA3()

Dieser Testfall überdeckt folgende Transitionen:

tA1, tB1, tA3, tB2, tA5, <u>tA6, tB4</u>, tB5, tB2, tA5, <u>tA6, tB3</u>.

4.3.4. Genetische Operatoren

Um aus einer existierenden Population eine neue Population zu generieren, müssen genetische Operatoren wiederholt angewendet werden. Sie werden so lange auf die Individuen der existierenden Population angewendet, bis in der neuen Population genügend Individuen vorhanden sind, wobei die Populationsgröße vor dem Start der Generierung festgelegt wird. Vor der ersten Anwendung der Operatoren werden die Individuen der aktuellen Population ihrer Fitness nach absteigend sortiert, so dass auf Position 0 der Population das fitteste Individuum und auf Position *Populationsgröße-1* das Individuum mit der schlechtesten Fitness ist.

4.3.4.1. Elitismusoperator

Als erstes wird ein Elitismusoperator angewendet, um sicherzustellen, dass sich bei der Erzeugung neuer Populationen keine Verschlechterung im Vergleich zu der alten Population ergibt. Deshalb werden die besten Individuen der aktuellen Population unverändert in die neue Population übernommen, indem die Individuen beginnend von der Position 0 ausgewählt werden. Der Anteil an unverändert zu übernehmenden Individuen wird vor dem Start der Generierung festgelegt. Dies wird als prozentueller Anteil bezüglich der Populationsgröße angegeben.

Beispiel Bei einer Populationsgröße von 100 Individuen und einer 5%igen Elitismuseinstellung, werden die Individuen der aktuellen und sortierten Population mit Index 0 bis 4 unverändert übernommen.

4.3.4.2. Selektionsoperator

Um zu gewährleisten, dass hauptsächlich sehr gute Individuen ausgewählt werden, schlechte Individuen aber auch eine Chance bekommen, generiert der umgesetzte Operator bei jeder An-

wendung zwei Zufallszahlen aus dem Intervall [-1, 1)[15]. Die generierten Werte werden mit der Populationsgröße multipliziert und die absoluten Beträge dieser Zahlen gespeichert. Diese zwei Zahlen stellen die Indizes der Individuen aus der aktuellen Population dar, die ausgewählt werden.

Beispiel Anhand eines Experiments wird gezeigt, dass die generierten Zahlen meistens sehr klein sind. Es wurde ein Generierungsvorgang mit einer Populationsgrößeneinstellung von 100 gestartet und für 10000 Selektionsschritte festgehalten, welche Indizes ausgewählt wurden. In jedem Selektionsschritt hat der Index einen ganzen Wert zwischen 0 und 99, weil die Populationsgröße 100 ausgewählt wurde. Abbildung 4.3 zeigt wie oft Indizes aus bestimmten Wertebereichen ausgewählt wurden: der Bereich 0–10 stellt die Anzahl aller Indizes dar, die größer oder gleich 0 sind und kleiner als 10; der Bereich 10–20 stellt die Anzahl aller Indizes dar, die größer oder gleich 10 sind und kleiner als 20 usw.

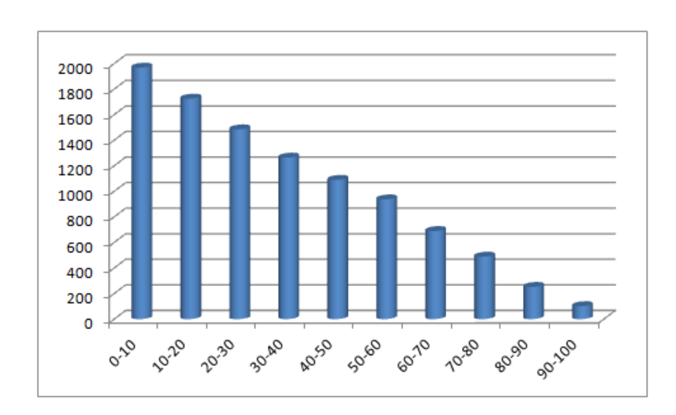


Abbildung 4.3.: Ergebnisse eines Experiments zur Veranschaulichung der Funktionsweise des Selektionsoperators

Durch diesen Selektionsoperator werden also hauptsächlich Individuen ausgewählt, die auf den ersten Stellen der aktuellen Population stehen. Da diese Population der Fitness nach absteigend sortiert ist, handelt es sind dabei um die besten Individuen der aktuellen Population. D. h., diese Strategie ist vergleichbar mit der in Abschnitt 4.2 beschriebene *Roulette-wheel-selection*,

[15]Um einen solchen Wert zu generieren wird als erstes die Funktion nextDouble() der Java-Math Klasse aufgerufen, die eine gleichverteilte Zufallszahl *randomValue* mit Werten aus dem Intervall [0.0d, 1.0d) zurückgibt. Falls der Wert in *randomValue* kleiner als 0,5 ist, wird der Wert *Math.sqrt(2 * randomValue) - 1* generiert, anderenfalls der Wert *1 - Math.sqrt(2 - 2 * randomValue)*

wobei die Wahrscheinlichkeit nicht proportional zu der Fitness ist. Aus diesen zwei ausgewählten Individuen werden durch den Rekombinationsoperator neue Individuen erstellt.

4.3.4.3. Rekombinationsoperatoren

Der umgesetzte Rekombinationsoperator erzeugt durch die Kombination zweier Eltern-Individuen immer zwei Kind-Individuen. Beide erzeugten Individuen sind dann Teil der neu generierten Population. Im Allgemeinen geht es bei der Rekombination darum, Eigenschaften zwischen den zwei Eltern-Individuen auszutauschen. Dabei gibt es mehrere Teile von Individuen, die durch Rekombination zwischen diesen ausgetauscht werden können. Die Rekombination kann also auf mehreren Ebenen stattfinden:

- *Testfälle*: ganze Testfällen können zwischen Individuen ausgetauscht werden.
- *Aufrufe*: einzelne Aufrufe aus den Testfällen können zwischen den Individuen ausgetauscht werden.
- *Testdaten*: die Daten einzelner Aufrufe (die Werte einzelner Parameter von Aufrufen) aus den Testfällen der Individuen können mittels dieses Operators kombiniert[16] werden. Dies funktioniert natürlich nur für Parameter, die den gleichen Typ haben.

Auf welcher Ebene die Rekombination tatsächlich stattfindet wird bei jeder Anwendung des Rekombinationsoperators aus diesen drei Alternativen *zufällig ausgewählt*. Diese drei Alternativen werden in Abbildung 4.4 bildlich dargestellt.

```
opA1(-1104924320), opB4(), opA5()
opA1(1456540800), opB4(), opA2(), opA3(), opA2(), opA3(), opA4(true), opA5()
opA1(-129423737), opB4(), opA2(), opA3(), opA4(true), opA5()
opA1(-1787143425), opB4(), opA5()
opA5(), opB4()

opA1(1500943729), opB1(-2050836360), opA5(), opB4()
opA5(), opB4()
opA5(), opB1(-1555817572), opB4()
opA5(), opB1(1329393341), opB4()
.
.
.
opA1(-1924151534), opB1(2076310642), opA2(), opA3(), opB4(), opA2(), opA3(), opA2(), opA3(), opA2(), opA3()
opA1(1770596331), opB1(1627284803), opA5(), opB4()
opA1(-1988063922), opB4(), opA2(), opA3(), opA2(), opA3(), opA2(), opA3(), opA2(), opA3()
opA1(1794321776), opB1(1052368042), opA5(), opB4()
opA5(), opB1(-620774538), opB4()
```
(a) (b) (c)

Abbildung 4.4.: Schemenhafte Darstellung der umgesetzten Rekombinationstypen:
(a): auf Testfallebene; **(b)**: auf Aufrufebene; **(c)**: auf Testdatenebene;

[16]Was das genau bedeutet, wird im Folgenden beschrieben

Für die *Rekombination auf Ebene der Testfälle oder der Aufrufe* wurden drei Rekombinationsoperatoren umgesetzt. Diese Operatoren unterscheiden sich in der Art und Weise wie sie Kreuzungspunkte ansetzen. Gemeinsam haben sie, dass sie auf Folgen oder Mengen von Entitäten angewendet werden können. D. h., diese Operatoren können sowohl auf Individuen (Menge von Testfällen) als auch auf einzelne Gene (Folge von Aufrufen) angewendet werden. Folgende Operatoren wurden umgesetzt:

- *SinglePoint*: es wird eine Stelle zufällig ausgewählt, ab dem die Elemente ausgetauscht werden.
- *TwoPoint*: es werden zwei Stellen ausgewählt. Zwischen der ersten und der zweiten Stelle werden zwischen den Individuen Elemente ausgetauscht.
- *Uniform*: es werden alle Elemente durchlaufen und jedes zweite Element zwischen den beiden Entitäten ausgetauscht.

Welcher dieser drei Ausprägungen bei der Kreuzung auf Testfall oder Aufrufebene im Einzelfall zum Einsatz kommt, wird bei jeder Anwendung des Rekombinationsoperators zufällig ausgewählt.

Für die *Rekombination auf Ebene der Daten* gibt es ebenfalls mehrere Ausprägungen. Diese haben gemeinsam, dass anhand der beiden Werte ein neuer Wert generiert wird, der dem Parameter aus dem ersten Individuum zugeordnet wird. Abhängig vom Typ der Daten gehen auch die Operatoren unterschiedlich vor. Hier werden nur die Operatoren beschrieben, die auf nummerischen Parametern agieren, da die Beschreibung der Operatoren für alle möglichen Datentypen den Rahmen sprengen würde.

Eine Ausprägung davon ist der *AverageOperator*, der den neuen Wert durch Mittelung berechnet. Falls der *RandomInterval*-Operator angewendet wird, wird der neue Wert zufällig aus dem Intervall zwischen den zwei betrachteten Werten ermittelt. Der *SinglePoint*-Operator betrachtet die Werte als Liste von Ziffern, setzt für jeden Wert einen zufälligen Kreuzungspunkt an und tauscht die Ziffern ab dem Kreuzungspunkt zwischen den Werten aus. Welcher dieser Ausprägungen bei der Kreuzung auf Datenebene im Einzelfall zum Einsatz kommt, wird aus diesen drei Alternativen zufällig ausgewählt.

Bei jeder Anwendung des Rekombinationsoperators werden also zwei zufällige Entscheidungen getroffen. Erstens auf welcher Ebene die Rekombination stattfinden soll. Abhängig von der getroffenen Auswahl wird in einem zweiten Schritt zufällig entschieden, welche konkrete Aus-

prägung des Operators zum Einsatz kommt.

4.3.4.4. Mutationsoperatoren

Nachdem durch Rekombination zwei neue Individuen entstanden sind, wird jedes Individuum mit einer vor dem Start der Generierung festgelegten Wahrscheinlichkeit mutiert. Darüber hinaus beschreibt ein zusätzlicher Parameter (die Mutationsvarianz), wie stark die durchgeführten Änderungen sein dürfen.

Ähnlich wie die Rekombinationsoperatoren können auch die Mutationsoperatoren auf verschiedene Eigenschaften der Individuen angewendet werden. Auf welcher Ebene die Mutation stattfindet, wird vor jeder Mutation zufällig entschieden, dabei gibt es drei Möglichkeiten:

- *Testfälle*: Bei Anwendung dieser Mutationsart werden dem Individuum neue Testfälle hinzugefügt oder bestehende Testfälle gelöscht, wobei zufällig entschieden wird, ob hinzugefügt oder gelöscht werden soll. Wie viele Testfälle gelöscht oder hinzugefügt werden, hängt von der Mutationsvarianz ab, es handelt sich aber immer um mindestens einen Testfall.
- *Aufrufe*: Analog geht der Operator auf Ebene der Aufrufe vor, mit dem Unterschied, dass nun nicht ganze Testfälle hinzugefügt oder gelöscht werden, sondern nur einzelne Aufrufe.
- *Testdaten*: Da für einen bestehenden Aufruf nicht einfach Daten (also Parameterwerte) gelöscht oder hinzugefügt werden dürfen (sonst wäre der Aufruf unvollständig), geht der Operator wie folgt vor: zunächst wird zufällig ein Aufruf ausgewählt, der einen Parameter enthält. Abhängig von dem Typ des Parameters wird dann eine entsprechende Mutation umgesetzt. So z. B. wird, falls es sich um einen nummerischen Wert handelt, dieser Wert um einen zufällig bestimmten Wert[17] erhöht oder verkleinert.

4.3.5. Abbruchbedingungen

Der genetische Algorithmus generiert durch Anwendung der oben beschriebenen Operatoren immer neue Populationen, bis eine – vor dem Start der Generierung einzustellende – Abbruchbedingung erfüllt ist. Die im Folgenden beschriebenen Abbruchbedingungen geben an, wann die

[17] Dieser Wert hängt auch von der Mutationsvarianz ab

globale Optimierung[18] angehalten wird:

- *keine Abbruchbedingung*: in diesem Fall generiert der Algorithmus so lange Testfälle, bis der Generierungsprozess manuell durch den Benutzer angehalten wird.
- *maximale Anzahl an Generationen*: falls diese Abbruchbedingung eingestellt ist, wird der Generierungsprozess beendet, nachdem eine bestimmte Anzahl an Generationen erzeugt wurde.
- *gewünschte Überdeckung erreicht*: für den Fall, dass diese Abbruchbedingung eingestellt ist, beendet der Algorithmus die Generierung, falls ein generiertes Individuum die gewünschte Überdeckung erreicht hat.

[18] Die Abbruchbedingungen der *lokalen Optimierung* wurden in Abschnitt 4.3.1 beschrieben

5. Werkzeugunterstützung

Dieses Kapitel beschreibt das implementierte Werkzeug, das mittels der im vorigen Kapitel beschriebenen genetischen Algorithmen Testfälle vollautomatisch erzeugt. In Abschnitt 5.1 wird der Aufbau der graphischen Benutzeroberfläche des Werkzeugs zusammen mit den dadurch zur Verfügung gestellten Funktionalitäten beschrieben. Anschließend werden in den darauf folgenden Abschnitten weitere nützliche Funktionalitäten des Werkzeugs vorgestellt, und zwar die Visualisierung der erreichten Modellüberdeckung (Abschnitt 5.2) und die Regressionstestgenerierung (Abschnitt 5.3). Eine weitere wichtige Funktionalität des Werkzeugs ist die Unterstützung des Modell-Mutationstests. Diese wird allerdings erst in Kapitel 7 vorgestellt.

5.1. Das Werkzeug UnITeD

Das Werkzeug namens UnITeD wurde in der Programmiersprache Java umgesetzt und erlaubt die vollautomatische Testfallgenerierung (Testsequenzen inklusive zugehöriger Testdaten) von modellbasierten Komponenten- und Integrationstestfällen. Bei der Entwicklung kam dabei die Open-Source Softwareentwicklungsplattform *Eclipse*[1] zum Einsatz. Für die Versionsverwaltung der entstehenden Implementierungsartefakte wurde das Open-Source Eclipse-Plugin *Subversion*[2] benutzt. Die Bedienung dieses Werkzeugs wird in Anhang B beschrieben.

Das Werkzeug UnITeD kann mit Modellen umgehen, die mittels unterschiedlicher UML-Werkzeuge erstellt wurden. Voraussetzung dafür ist, dass die Werkzeuge den Export in das *EMF UML2 XMI*-Format unterstützen. Hier kann von den Erfahrungen mit der Benutzung von zwei UML-Werkzeugen berichtet werden. Zum einen handelt es sich um das Werkzeug *MagicDraw*[3]

[1] http://www.eclipse.org/eclipse/
[2] http://subversion.tigris.org/
[3] Die benutzte in Version ist: 15.5

und zum anderen um das Werkzeug *Enterprise Architect*[4]. Der von MagicDraw erzeugte Modellexport kann unverändert mit dem Werkzeug UnITeD geladen werden. Im Gegensatz dazu, ist der von Enterprise Architect erzeugte XMI-Export nicht ganz im vom Werkzeug UnITeD benötigten Format. Deshalb müssen diese XMI-Dateien mittels eines *XSLT*[5]-Skripts nachträglich bearbeitet werden.

Neben der Beschreibung der statischen Struktur und des dynamischen Verhaltens des zu modellierenden Systems, muss das Modell auch eine *Testsystembeschreibung* enthalten. Diese beinhaltet zusätzliche Informationen, die für die Testfallerstellung relevant sind. Dazu enthält es mehrere *TestContexte*[6], die eine oder mehrere Komponenten referenzieren. Darüber hinaus enthält jeder *TestContext* ein *Setup*-Sequenzdiagramm, das die referenzierte(n) Komponente(n) instanziiert und Beziehungen zwischen ihnen herstellt.

Da ein Modell im Allgemeinen mehrere Komponenten enthalten kann, wird dieses Testsystem benötigt, um bei der Testfallgenerierung den Fokus auf bestimmte Komponenten dieses Modells einschränken zu können. Dazu muss beim Start der Testfallgenerierung ein bestimmter *TestContext* ausgewählt werden. Der Dialog, der dies ermöglicht wird in Abbildung 5.1 dargestellt.

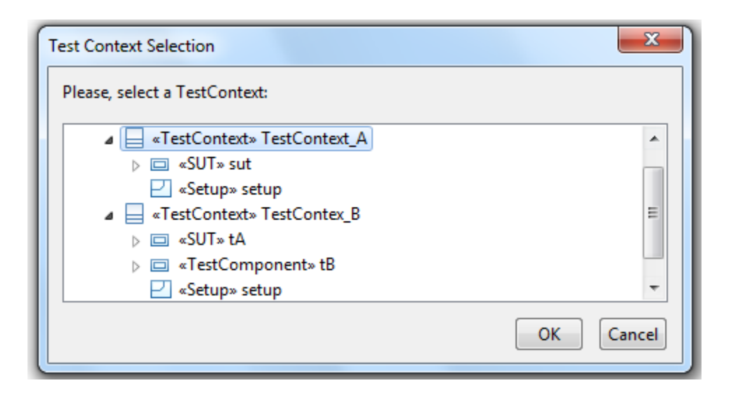


Abbildung 5.1.: Screenshot UnITeD: Dialog zur Auswahl eines TestContextes

Wenn beim Start des Generierungsprozesses ein TestContext ausgewählt wird, der nur eine Komponente referenziert[7], dann wird nur der Zustandsautomat dieser Komponente berücksichtigt. Das bedeutet, dass die generierten Testfälle nur den einen Zustandsautomaten überdecken. Ein Beispiel für einen TestContext, der nur eine Komponente referenziert ist der *TestContext_A*

[4]Die benutzte Version ist: 7.0
[5]Kurz für Extensible Stylesheet Language Transformations
[6]Sind Klassen, die den Stereotyp «TestContext» haben
[7]D.h., dieser TestContext enthält eine Variable vom Typ der Komponente

aus Abbildung 5.1. Dieser enthält mit der Variable *sut* eine Referenz auf die Komponente *A*[8].

Für den Fall, dass ein TestContext bei der Generierung ausgewählt wird, der mehrere Komponenten referenziert, zielt die Testfallgenerierung auf die Überdeckung der Interaktionen der referenzierten Komponenten. Ein Beispiel für so einen TestContext ist der *TestContext_B* aus Abbildung 5.1. Dieser enthält durch die Variable *tA* eine Referenz auf Komponente *A* und durch die Variable *tB* eine Referenz auf Komponente *B*[9]. Dabei wird der Variable *tA* der Stereotyp «SUT» zugewiesen und der Variable *tB* der Stereotyp «TestComponent». Dies beruht auf der im Rahmen des Projekts gewählten Modellierungskonvention: falls nur eine Komponente referenziert wird, bekommt die entsprechende Variable den Stereotyp «SUT». Falls mehrere Komponenten referenziert werden, muss eine Variable mit dem Stereotyp «SUT» belegt werden und alle anderen Variablen mit dem Stereotyp «TestComponent». Für die Testfallgenerierung ist der Stereotyp der einzelnen Variablen nicht relevant, es ist nur wichtig, dass alle Variablen mit einem Stereotyp versehen werden[10].

Nachdem die vom Werkzeug UnITeD benötigten Eingaben beschrieben wurden, werden im folgenden Teil dieses Abschnittes die Funktionalitäten des Werkzeugs beschrieben. Diese werden über eine GUI zur Verfügung gestellt, die in Abbildung 5.2 gezeigt[11] und im Folgenden kurz beschrieben wird.

Im Bereich *Modellbaum* wird in einer Baumdarstellung das geladene UML-Modell[12] dargestellt, das sowohl das System als auch die Testsystembeschreibung mit den entsprechenden TestContexten enthält. Links unten werden im Bereich *Eigenschaften eines ausgewählten Modellbaumelements* die Eigenschaften eines ausgewählten Elements des Modellbaums dargestellt.

Rechts oben werden im Bereich *Generierte Testsuites* die aus dem Modell generierten Testsuites dargestellt. Aus dieser Liste kann durch Anklicken eine Testsuite auswählt werden, was zur Folge hat, dass im Bereich *Eigenschaften einer ausgewählten Testsuite* die Eigenschaften der ausgewählten Testsuite dargestellt werden. Dieser Bereich enthält mehrere Informationen. Zum einen wird beschrieben, für welchen TestContext diese Testsuite generiert wurde. In diesem Fall handelt es sich um den *TestContext_B*; die Testsuite wurde also generiert, mit dem Ziel die Map-

[8]Der Zustandsautomat dieser Komponente ist in Abbildung 3.1 dargestellt

[9]Der Zustandsautomat dieser Komponente ist in Abbildung 3.2 dargestellt

[10]D. h., man könnte genauso gut *tA* den Stereotyp «TestComponent» zuordnen und *tB* den Stereotyp «SUT»

[11]Dieser Screenshot wurde bearbeitet, in dem einzelne Bereiche der GUI markiert und benannt wurden

[12]In diesem Fall handelt es sich um ein Modell, das in MagicDraw erstellt wurde und die zwei Komponenten aus Abbildung 3.2 enthält

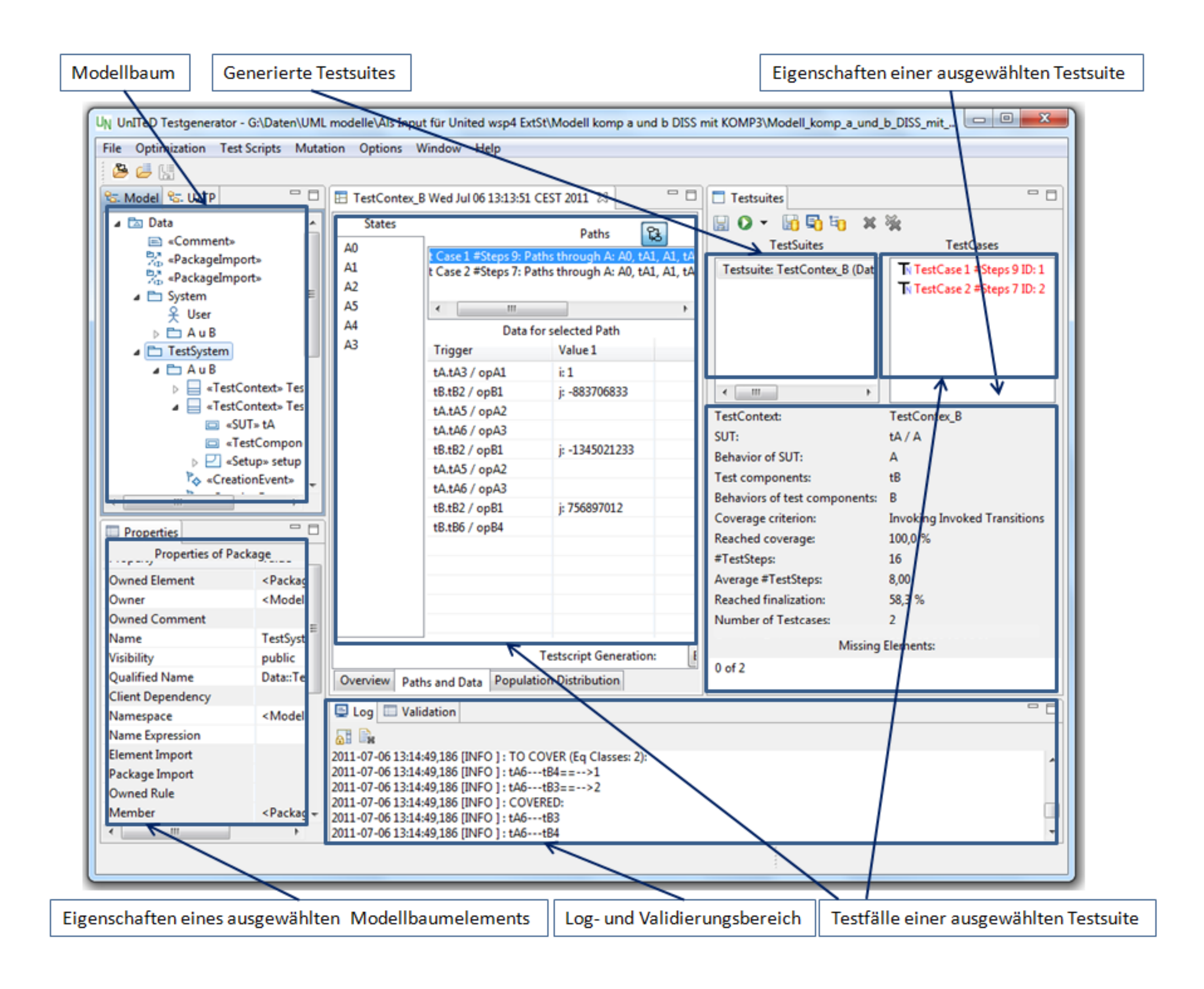


Abbildung 5.2.: Screenshot UnITeD: Aufbau der Benutzeroberfläche

pings zwischen den zwei Komponenten *A* und *B* zu überdecken. Darunter wird das anvisierte Überdeckungskriterium beschrieben, zusammen mit der von der Testsuite erreichten Überdeckung. Des Weiteren wird auch die Anzahl an Teststeps (= Anzahl an Aufrufen) angegeben, die als Summe aller Aufrufe über alle Testfälle der Testsuite berechnet wird. Weiter unten wird die mittlere Anzahl an Teststeps pro Testfall angegeben, gefolgt von dem Anteil an finalisierten Testfällen[13]. Schließlich wird die Anzahl der Testfälle angegeben, die in der Testfallmenge enthalten sind. Unter Missing Elements, werden die Entitäten angegeben, die zu überdecken waren, aber mittels der betrachteten Testsuite nicht überdeckt werden konnten. Abhängig vom Überdeckungskriterium werden hier also einzelne Zustände, Transitionen, Transitionspaare oder Mappings angegeben. Durch den ausgewählten Testfall wurden beide Mappings überdeckt (*„Reached Coverage : 100%“* in Abbildung 5.2), was zur Folge hat, dass die Liste der Missing Elements leer ist.

Durch das Auswählen einer Testsuite werden zusätzlich in den zwei Bereichen, die von *Testfälle einer ausgewählten Testsuite* referenziert werden, die darin enthaltenen Testfälle dargestellt. Im rechten Bereich werden die Testfälle nur als Liste übersichtlich dargestellt, wohingegen im linken Bereich die genauere Betrachtung einzelner Testfälle ermöglicht wird. Hier hat die Auswahl eines spezifischen Testfalls zur Folge, dass die einzelnen Aufrufe des Testfalls dargestellt werden. Während der Generierung wird in diesem Fenster der Generierungsfortschritt angezeigt.

Durch Rechtsklick mit der Maus auf eine Testsuite im Bereich *Generierte Testsuites* erscheint ein Kontextmenü, das mehrere Funktionalitäten zur Verfügung stellt, die auf eine Testsuite bezogen sind. Zum einen kann der Generierungsprozess ausgehend von der ausgewählten Testsuite weitergeführt werden. In diesem Fall wird diese Testsuite als Basis für die Generierung der Anfangspopulation genommen. Dies kann dann von Interesse sein, wenn die generierte Testsuite noch nicht die gewünschte Überdeckung erreicht hat oder eine weitere Optimierung bezüglich der Anzahl enthaltener Testfälle erwünscht ist. Des Weiteren können über dieses Kontextmenü Testsuites gelöscht werden. Darüber hinaus bietet dieses Menü die Möglichkeit Visualisierungsinformationen abzuspeichern, die dann in ein Modellierungswerkzeug eingelesen werden können, um die Testfallüberdeckung zu veranschaulichen. Detailliert wird die Visualisierung in Abschnitt 5.2 beschrieben.

Im Bereich *Log- und Validierungsbereich* stehen zwei Reiter zur Verfügung. Ist der Reiter

[13]Ein Testfall ist finalisiert, wenn er komplette Pfade (vom Startzustand zum Endzustand) durch Zustandsautomaten beschreibt

Log ausgewählt, werden während der Testfallgenerierung Informationen zum Generierungsfortschritt beschrieben. Diese Informationen beinhalten Angaben zu den bisher generierten Populationen und den besten darin enthaltenen Individuen. Der Reiter Validation beschreibt Probleme im geladenen Modell. Ein Beispiel für solch ein Problem ist, dass im TestContext keine Variable definiert ist, die den Stereotyp «SUT» hat. Des Weiteren werden auch Probleme in den einzelnen Automaten aufgezeigt, z. B. wenn sich ein Guard auf Variablen bezieht, die nicht definiert sind[14].

Die Hauptfunktionalitäten des Werkzeugs werden über das Menü aufgerufen. Der Menüpunkt *File* stellt Funktionalitäten zum Laden/Entladen des Modells und zum Speichern oder Löschen der generierten Testsuites zu Verfügung.

Der Menüpunkt *Optimization* erlaubt es, einen neuen Generierungsvorgang zu starten. Das Starten der Generierung erlaubt zunächst das Auswählen eines TestContext (siehe Abbildung 5.1); abhängig von dem ausgewählten TestContext werden danach Komponenten- oder Integrationstestfälle generiert. Vor dem Start der Generierung kann der Konfigurationsdialog aufgerufen werden, der es ermöglicht diverse, für die Generierung wichtige Einstellungen zu ändern. Zu diesen Einstellungen zählen:

- das anvisierte Überdeckungskriterium auf Komponenten- und auf Integrationsebene
- die Parametrisierung des Algorithmus
- die Gewichtung der einzelnen Fitnesswerte
- die Abbruchbedingung

Diese Einstellungen werden abgespeichert und müssen dann nicht vor jedem Generierungsprozess neu gesetzt werden. Vor dem ersten Starten des Werkzeugs sind noch die Standardeinstellungen abgespeichert, die jederzeit wieder hergestellt werden können.

Der Menüpunkt *Test Skript* ermöglicht es, die generierten Testfälle in diverse Formate zu exportieren. Unter anderem werden sie in ein Format exportiert, das als Eingabe für das Testmanagement-Werkzeug *HP TestDirector* benutzt werden kann.

Der Menüpunkt *Mutation* erlaubt das Aufrufen des Moduls, das die Bewertung der generierten Testsuites hinsichtlich deren Potential Modellierungsfehler zu entdecken[15] ermöglicht.

Unter dem Menüpunkt *Options* können diverse Einstellungen vorgenommen werden, z. B. ob

[14] Die zulässige Modellierung von Bedingungen in Guards wurde in Abschnitt 3.1 beschrieben
[15] Siehe Kapitel 7

der Menüpunkt *Mutation* überhaupt angezeigt werden soll[16] oder ob die Log-Informationen[17] im Log-Fenster angezeigt werden sollen. Schließlich ermöglicht der Menüpunkt *Window* das Verwalten der einzelnen Bereiche der GUI. Der Menüpunkt *Help* ermöglicht das Abrufen eines About-Dialogs, der diverse Informationen zum Werkzeug UnITeD darstellt, z. B. auf welchen Plugins dieses Werkzeug basiert.

Die gerade beschriebene GUI ermöglicht es also den Testfallgenerierungsprozess zu starten und die generierten Testsuites zu verwalten. Die bei der Generierung von modellbasierten Komponententestfällen bzw. von Integrationstestfällen erzielten Ergebnisse werden in Kapitel 6 beschrieben. Davor werden in den nächsten zwei Abschnitten weitere nützliche Funktionalitäten des Werkzeugs beschrieben, und zwar die Visualisierung der Modellüberdeckung und die Regressionstestgenerierung.

5.2. Visualisierung überdeckter sowie zu überdeckender Modellelemente

Neben der Testfallgenerierung erlaubt das vorgestellte Werkzeug auch die Visualisierung der durch generierte Testsuites überdeckten Modellentitäten. Diese Funktionalität wurde im Rahmen einer Diplomarbeit [Neu08] am Lehrstuhl für Software Engineering in Erlangen umgesetzt und im Rahmen einer Publikation ([PSN09]) erstmalig vorgestellt.

Die Visualisierung der erreichten Überdeckung ist in mehrfacher Hinsicht hilfreich [PSN09]:

- Sie unterstützt die *Testfallbewertung*: durch Veranschaulichung der durch eine Testsuite überdeckten Modellentitäten wird die Bewertung der vorliegenden Testsuite durch den Benutzer unterstützt. Dies ist besonders dann wichtig, wenn die gewünschte Abdeckung durch die generierte Testsuite nicht erreicht wurde. In diesem Fall schafft die Visualisierung eine Entscheidungsgrundlage, die es dem Benutzer ermöglicht, entsprechende Maßnahmen zu treffen. Z. B. kann der Benutzer dann durch Kenntnis der nicht überdeckten Entitäten, manuell Testfälle generieren, die diese Entitäten überdecken. Auch im Falle der Er-

[16] Diese Funktionalität wird bei der Evaluierung zu akademischen Zwecken benutzt; Personen, die das Werkzeug zur Testfallgenerierung im industriellen Kontext einsetzen, benötigen diese Funktionalität wahrscheinlich nicht

[17] Dies sind textuelle Informationen zu dem Fortschritt der Testfallgenerierung und beinhalten die Anzahl generierter Populationen und die erreichte Überdeckung

zielung einer 100%igen Überdeckung unterstützt die Visualisierung den Benutzer. Durch Visualisierung der von einem einzigen Testfall überdeckten Entitäten, können die Testfälle leicht identifiziert werden, die bestimmte Modellteile überdecken. Dies ist hilfreich, falls nur ein Testfall ausgeführt werden soll, der einen spezifischen Teil des Modells überdeckt.

- Sie unterstützt die *Bewertung der Modelle*: Die Visualisierung der überdeckten Entitäten eignet sich auch für die Überprüfung der Modelle, die der Testfallgenerierung zugrunde liegen. Zur Überprüfung solcher Modelle haben sich Methoden wie Inspections oder Walkthroughs[18] durchgesetzt. Die manuelle Natur solcher Verfahren birgt aber die Gefahr in sich, dass dabei bestimmte Teile des Modells übersehen werden. Gegenüber diesen Methoden hat die Anwendung modellbasierter Testfälle den Vorteil der systematischen Abdeckung aller Modellentitäten. Die Visualisierung einzelner Testfälle im Diagramm unterstützt die Erkennung *fehlerhafter Übergänge* oder *fehlerhafter Pfade*. Die Anzeige der nicht überdeckten Modellelemente bietet darüber hinaus eine Hilfestellung beim Finden von Modellfehlern, die auf nicht ausführbare Modellteile zurückzuführen sind.

Die in der vorliegenden Arbeit betrachteten Kriterien sind auf die strukturelle Überdeckung von Modellen ausgerichtet, weshalb sie verlangen, dass spezifische Entitäten der Modelle überdeckt werden. Diese Entitäten lassen sich wie folgt in Kategorien einteilen [PSN09]:

- *Atomare Entitäten*: sind elementare Bestandteile des Zustandsautomaten, also Zustände und Transitionen.
- *Zusammengesetzte Entitäten*: bestehen aus mehreren atomaren Entitäten. Zur Überdeckung solcher Entitäten reicht die Überdeckung der einzelnen Bestandteile nicht aus, da diese Bestandteile im Zusammenhang überdeckt werden müssen. Zu den zusammengesetzten Entitäten gehören Transitionspaare, Mappings und Mapping-Gruppen.

Entsprechend dieser Aufteilung wird in den folgenden zwei Unterabschnitten beschrieben, wie die Visualisierung atomarer und zusammengesetzter Entitäten realisiert wurde.

[18] Siehe Abschnitt 2.4

5.2.1. Visualisierung überdeckter und noch zu überdeckender atomarer Entitäten

Die Visualisierung überdeckter atomarer Entitäten wird durch Färbung realisiert, indem die einzelnen Transitionen oder Zustände wie folgt eingefärbt werden:

- *Grün*, wenn sie von mindestens einem Testfall der Testfallmenge überdeckt wurden.
- *Rot*, wenn sie noch nicht überdeckt wurden, aber zu überdecken sind.
- *Gar nicht markiert*, wenn sie noch nicht überdeckt wurden und nicht unbedingt zu überdecken sind (wie z. B. nicht durchlaufene Transitionen im Falle einer angestrebten Zustandsüberdeckung).

Abbildung 5.3 zeigt, die von einer Testsuite auf dem Zustandsautomat InfotainmentGui[19] erreichte Überdeckung bezüglich des Kriteriums Transitionsüberdeckung. Diese Testsuite überdeckt fast alle Transitionen. Die nicht überdeckte Transition wird rot markiert.

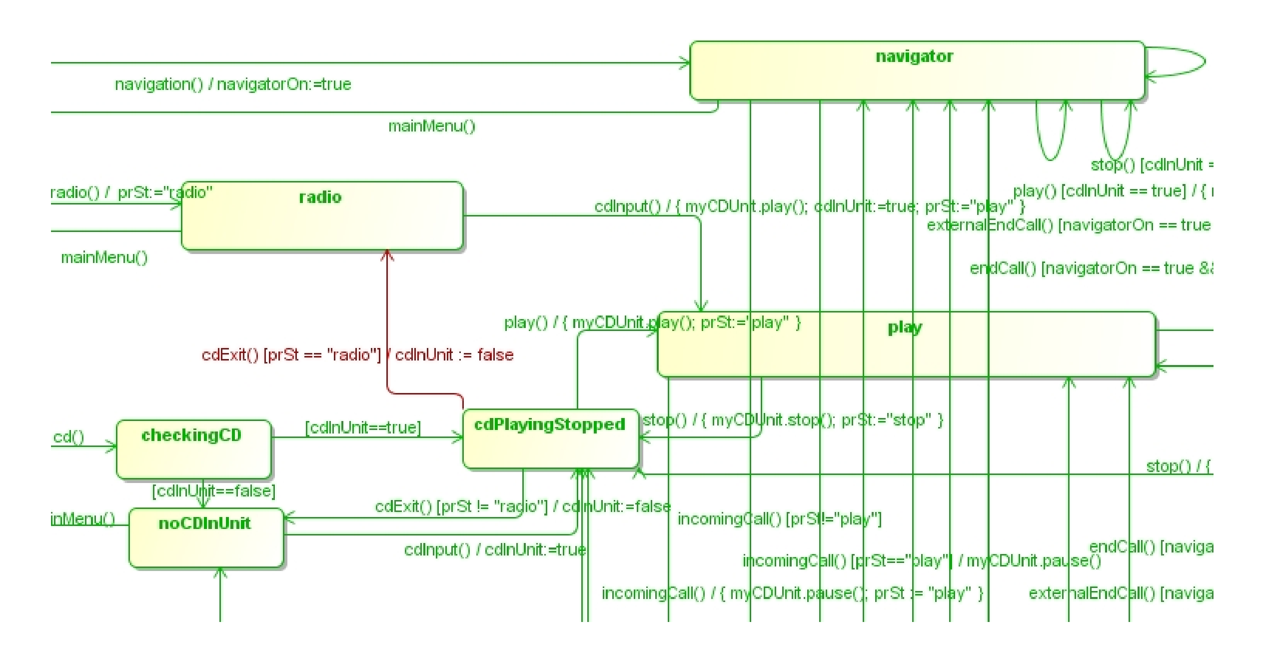


Abbildung 5.3.: Ausschnitt eines Zustandsautomaten, der um Überdeckungsinformationen einer Testsuite für den Komponententest ergänzt ist

[19]Details zu diesem Automat werden in Abschnitt 6.1 vorgestellt

Da neben der Visualisierung der Gesamtüberdeckung auch die Visualisierung der von einem Testfall erreichten Überdeckung interessant ist, wird auch die Farbe *Blau* benutzt. Zustände oder Transitionen werden so markiert, falls ein einzelner Testfall der Testsuite ausgewählt wird, für den die Überdeckung beschrieben werden soll. Abbildung 5.4 zeigt, die von einem Testfall (aus der vorhin erwähnten Testsuite) auf dem Zustandsautomat InfotainmentGui erreichte Überdeckung. Für die anderen Entitäten, die nicht von diesem Testfall überdeckt werden, bleibt die Färbung gleich.

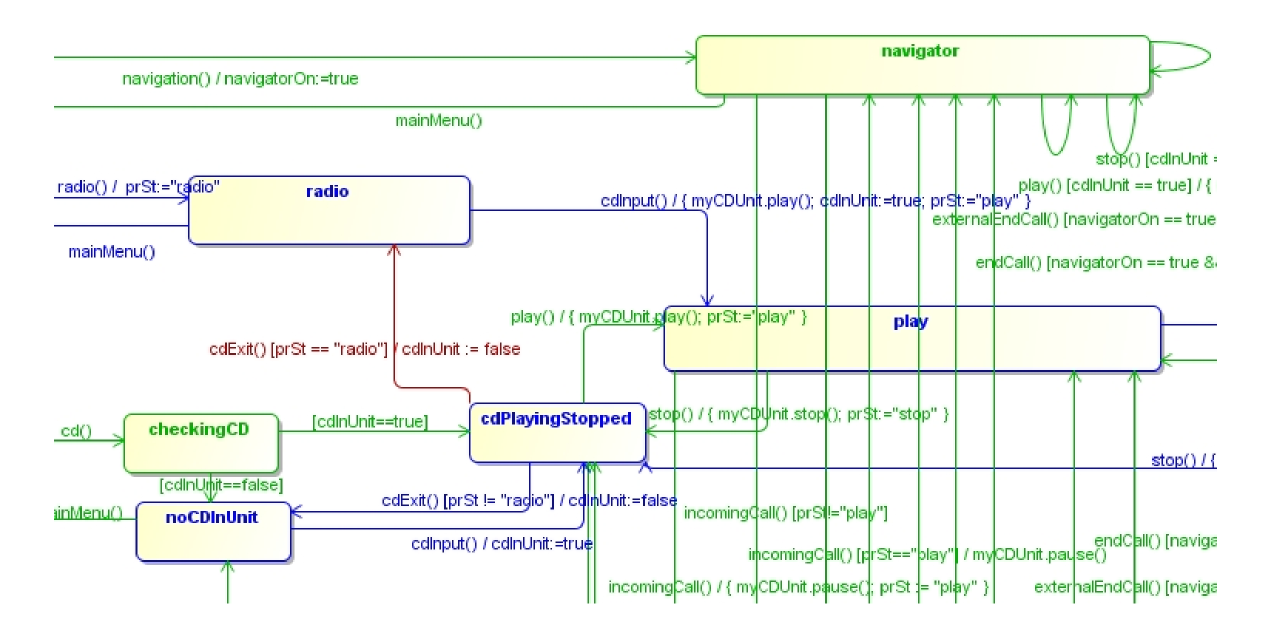


Abbildung 5.4.: Ausschnitt eines Zustandsautomaten, der um Überdeckungsinformationen eines einzelnen Komponententestfalls ergänzt ist

5.2.2. Visualisierung überdeckter und noch zu überdeckender zusammengesetzter Entitäten

Zur Visualisierung zusammengesetzter Entitäten reicht allein die Färbung nicht mehr aus, da diese Art von Entitäten sich als Kombination einzelner atomarer Entitäten beschreiben lassen, die zum Teil verschiedenen Diagrammen angehören[20]. Deshalb werden zur Beschreibung der zu überdeckenden zusammengesetzten Entitäten *Notizen* an den Transitionen der Diagramme

[20] Z. B.: Mappings bestehen aus zwei Transitionen aus verschiedenen Automaten

hinzugefügt. Diese Notizen enthalten Informationen zu den Mapping-Gruppen, zu denen diese Transition gehört. Diese Gruppen werden im Notiztext wie folgt dargestellt: auf der ersten Zeile wird der Name der Transition beschrieben, wonach auf jeder Zeile jeweils eine Mapping-Gruppe beschrieben wird. In jeder Zeile stehen also zueinander alternativ zu überdeckende Mappings.

Beispielhaft wird die Darstellung der Mapping-Gruppen in Abbildung 5.5 dargestellt. Die Notiz (a) zeigt die durch das Kriterium *Invoking / invoked transitions* durchgeführte Partitionierung der Mappings. Dieses Kriterium verlangt also die Überdeckung beider Mappings, da die Mappings auf zwei Zeilen getrennt dargestellt werden. Darunter zeigt die Notiz (b) die vom Kriterium *Invoking transitions on pre-states* durchgeführte Partitionierung. Für die Erfüllung dieses Kriteriums reicht es, wenn eines der zwei Mappings überdeckt wird.

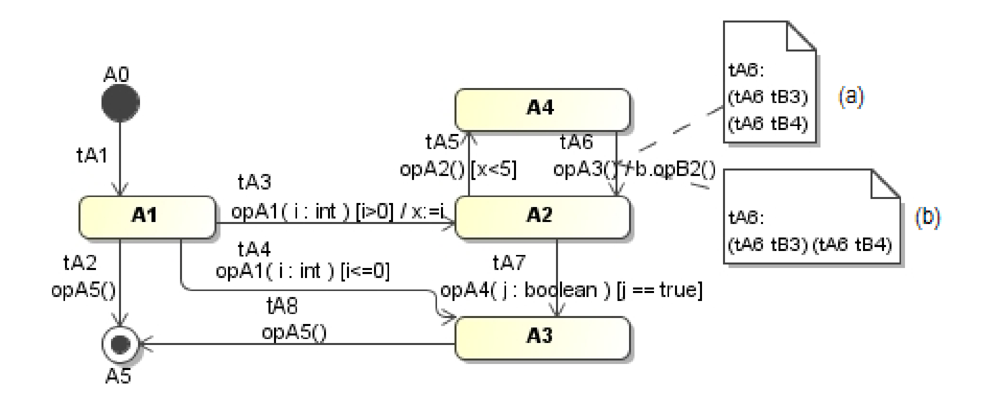


Abbildung 5.5.: Zustandsautomat ergänzt um Notizen mit Mappinginformationen [PSN09]

Nachdem die zu überdeckenden Mappings durch Notizen festgehalten wurden, kann innerhalb dieser Notizen nun markiert werden, welche Mappings durch eine Testsuite überdeckt wurden. Dies wird durch Änderung der Textfarbe in den Notiztextfeldern realisiert. Die benutzten Farben sind:

- *Grün*, wenn das Mapping von der Testsuite überdeckt wird.
- *Rot*, wenn es nicht überdeckt wird.

Nachdem Abbildung 5.5 zeigt, wie die zu überdeckenden Mappings festgehalten werden, beschreibt Abbildung 5.6, wie die überdeckten Mappings farblich markiert werden. Dazu wurde eine Testfallmenge aus diesem Modell generiert, die Visualisierungsinformationen abgespeichert und ins Modellierungswerkzeug geladen. Da durch die generierte Testsuite alle Mappings überdeckt wurden, gibt es in Abbildung 5.6 kein rot markiertes Mapping.

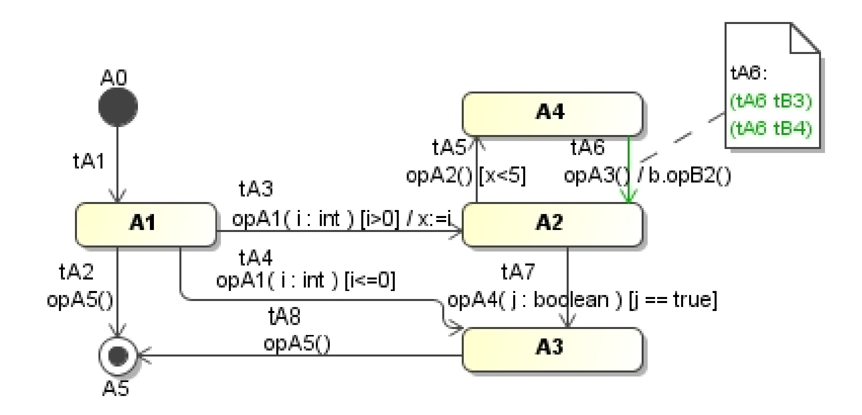


Abbildung 5.6.: Zustandsautomat ergänzt um Notizen und Informationen zu den überdeckten Mappings

Neben der Färbung der Notiztexte werden auch die Transitionen, an denen die Notizen angebracht sind, eingefärbt. Somit kann auch an der Farbe der Transition erkannt werden, inwieweit Mappings, zu denen diese Transition gehört, überdeckt wurden. Dies ist besonders im Falle sehr informationsdichter Notiztextfelder nützlich, da die Transitionsfärbung eine anschauliche Abstraktion darstellt. Die Transitionen werden wie folgt eingefärbt:

- *Grün*, wenn jede Mapping-Gruppe, zu der die Transition gehört, durch mindestens einen Testfall überdeckt wurde.
- *Rot*, wenn nicht alle Mapping-Gruppen, zu der die Transition gehört, von der Testfallmenge überdeckt wurden.

Zusätzlich wird auch die Farbe *Blau* wie folgt benutzt: ein blaues Mapping in einem Notiztextfeld bedeutet, dass dieses von einem bestimmten Testfall überdeckt wurde. Dagegen bedeutet die blaue Färbung der Transition, dass alle Mapping-Gruppen, zu der diese Transition gehört, durch den betrachteten Testfall überdeckt wurden. D. h., dass auf jeder Zeile der Notiz an der Transition mindestens ein blau markiertes Mapping vorhanden ist. Ein weiteres Beispiel, das auch die Benutzung der Farbe Blau illustriert wird in Abbildung 5.7 dargestellt.

Zur Implementierung der gerade beschriebenen Visualisierung wurde im Rahmen der Diplomarbeit [Neu08] ein Plugin für MagicDraw entwickelt, das es erlaubt, die von einer vorgegebenen Testsuite (oder einem darin enthaltenen Testfall) überdeckten Entitäten im Bezug auf ein auszuwählendes Überdeckungskriterium zu markieren. Einmal installiert, stellt das Plugin innerhalb von MagicDraw eine zusätzliche Toolbar zur Verfügung (siehe Abbildung 5.8), die es erlaubt für

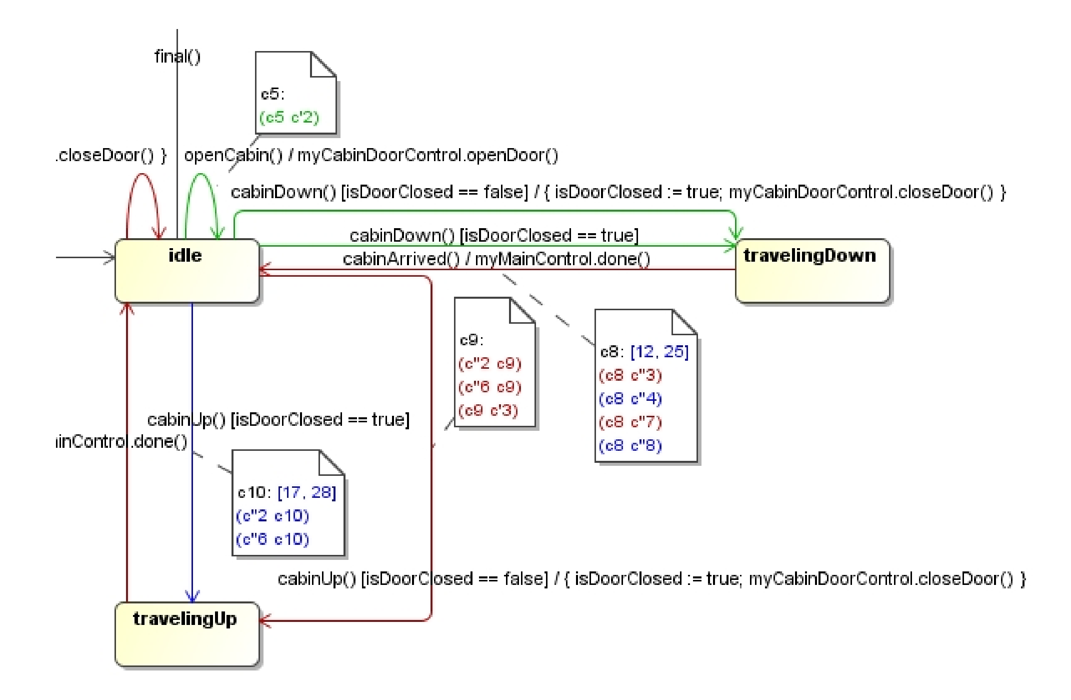


Abbildung 5.7.: Teil eines Zustandsautomaten, in dem Mappings und Transitionen farblich markiert sind

ein geladenes Modell eine mittels des Werkzeugs UnITeD gespeicherte Visualisierungsdatei[21] zu laden und zu entladen.

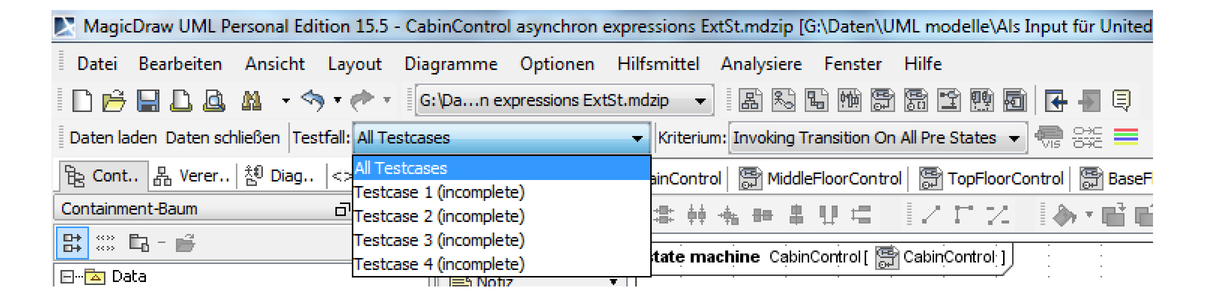


Abbildung 5.8.: Toolbar von MagicDraw

Darüber hinaus können in der Dropdown-Liste *Testfall* spezifische Testfälle dieser Testsuite ausgewählt werden. In der Dropdown-Liste *Kriterium* können spezifische Überdeckungskriterien ausgewählt werden. Für die ganze Testsuite oder einen ausgewählten Testfall wird dann die Überdeckung bezüglich des ausgewählten Kriteriums angezeigt.

[21] Siehe Abschnitt 5.1

5.3. Unterstützung des Regressionstests

Im Rahmen einer modellbasierten Softwareentwicklung sind Modelle die zentralen Entwicklungsartefakte und werden dementsprechend auch oft überarbeitet. Bei den dabei eingefügten Änderungen handelt es sich entweder um Korrekturen an vorhandenen Modellen oder um Erweiterungen und/oder Verfeinerungen der Modelle. Korrekturbedarf entsteht dann, wenn während der statischen Reviews oder der dynamischen Simulation von Modellen Fehler aufgedeckt werden. Verfeinerungen kommen dann vor, wenn man sich von der Ebene der abstrakten Spezifikationsmodelle immer weiter in Richtung detaillierter Entwurfs-Modelle bewegt. Unabhängig von der Art der Änderung ist es sinnvoll, bei Änderung eines Modells, die aus der vorherigen Version des Modells bereits generierten Testsuites weitgehend wiederzuverwenden, da für diese Testfälle das Testorakel schon definiert sein müsste. Für den Fall, dass vor Ausführung modellbasierter Testfälle eine Validierung dieser Testfälle durchgeführt werden muss, können durch die Wiederverwendung bereits validierter Testfälle Kosten eingespart werden.

Bei Änderungen des Modells ist es deshalb sinnvoll, bereits generierte Testfallmengen hinsichtlich ihrer Ausführbarkeit im geänderten Modell zu analysieren (*Regressionsanalyse*) und gegebenenfalls neue Testfälle hinzuzufügen (*Regressionstestgenerierung*). Neue Testfälle werden dann benötigt, wenn neue Elemente im Modell dazukommen oder bestimmte Elemente, die im alten Modell überdeckt wurden, im neuen Modell nicht mehr überdeckt werden (kann dann passieren, wenn bestimmte Testfälle auf dem neuen Modell nicht mehr lauffähig sind).

Mit der Fragestellung der *Regressionsanalyse* befassen sich eine Reihe von Forschungsarbeiten: [BLS02], [NR07] und [FIMN07]. Diese beschränken sich auf die Klassifikation bestehender modellbasierter Testfälle als *wieder verwendbare*, *neu zu validierende* und *zu verwerfende*. Darüber hinaus ermöglichen die Arbeiten von [CPU07] und [PUA06] die Generierung zusätzlich erforderlicher Testfälle, indem sie mittels statischer Analyse neue Pfade durch den Graphen ermitteln und somit neue Testsequenzen ableiten. Aufgrund der statischen Vorgehensweise stoßen diese Ansätze auch auf die in Abschnitt 4.1 schon angesprochenen Grenzen, die sich auf die Generierung von Pfaden beziehen, die eventuell gar nicht ausführbar sind.

Der hier beschriebene Ansatz (wurde erstmalig in [PBSO08] vorgestellt) zur Generierung zusätzlich erforderlicher Testfälle überwindet diese Grenzen durch die Benutzung heuristischer Verfahren und erlaubt darüber hinaus neben der Generierung passender Testdaten auch die Minimierung der Anzahl neu erforderlicher Testfälle.

Änderungen an Zustandsautomaten lassen sich wie folgt klassifizieren [PBSO08]:

- Erweiterung des Modells um zusätzliche graphische Elemente (Knoten oder Kanten).
- Entfernen graphischer Elemente.
- Umbenennung graphischer Elemente ohne Änderung ihrer Semantik (z. B. Änderung des Namens einer Transition. Dies ist keine Änderung der Semantik, da die Semantik einer Transition nur über die Elemente des 5-Tupels[22] definiert ist).
- Semantische Änderung graphischer Elemente (z. B.: Änderung des Namens eines Zustands. Im Gegensatz zur Änderung des Namens einer Transition ist die Änderung des Namens eines Zustands eine semantische Änderung).
- Verfeinerung von Modellteilen.

Basierend auf diesen Änderungsarten erlaubt das Werkzeug die bestehenden und neu generierten Testfälle wie folgt zu klassifizieren [PBSO08]:

- Klasse ***I**: Identical* beinhaltet Testfälle, die ausschließlich unveränderte Modellbereiche durchlaufen und deshalb unmittelbar übernommen werden können.
- Klasse ***X**: Obsolete* beinhaltet Testfälle, die im veränderten Modell nicht mehr lauffähig sind[23] und in ihrer ursprünglichen Form zu verwerfen sind.
- Klasse ***?**: Retest Required* beinhaltet Testfälle, die veränderte Modellbereiche durchlaufen, deren Verhalten deshalb neu überprüft werden muss.
- Klasse ***N**: New* beinhaltet neu hinzugefügte Testfälle.

Beispielhaft werden diese Konzepte an dem in Abbildung 5.9 dargestellten Zustandsautomaten veranschaulicht. Die Transition *t3* wurde entfernt, dafür eine neue Transition *t9* hinzugefügt. Der Name des Zustands *A5* wurde in *A5a* umgewandelt.

Vorausgesetzt es gibt eine Testsuite T ={T1, T2, T3}, die eine vollständige Transitionsüberdeckung der ursprünglichen Version dieses Automaten erreicht, indem die einzelnen Testfälle folgende Pfade durchlaufen:

T1: *A0, A1, A2, A3, A3, A6*
T2: *A0, A1, A4, A2, A3, A6*

[22] Siehe Abschnitt 3.1.2

[23] D. h., dass mindestens ein Aufruf des Testfalls bei der Simulation des Testfalls auf dem Zustandsautomaten nicht abgearbeitet werden kann

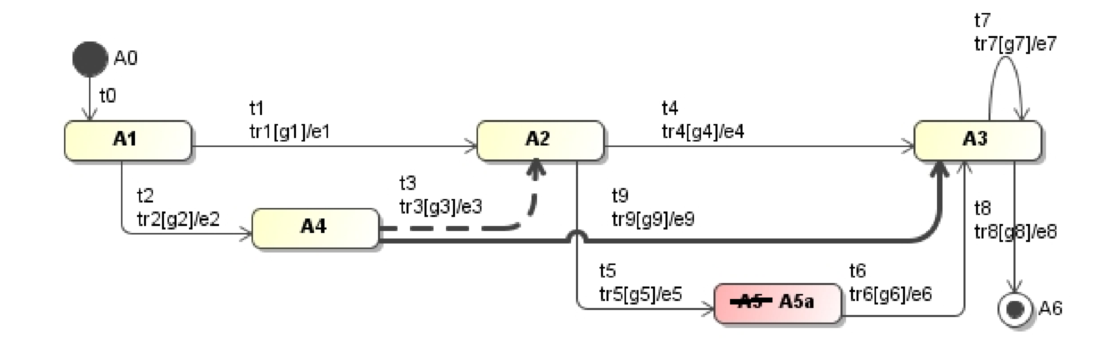


Abbildung 5.9.: Veränderter Zustandsautomat

T3: *A0, A1, A2, A5, A3, A6*

Im Hinblick auf die durchgeführten Änderungen am Automaten lassen sich die Testfälle der Testsuite T wie folgt klassifizieren:

- T1: gehört der Klasse *Identical (**I**)*, da er ausschließlich unveränderte Modellbereiche durchläuft und somit unverändert übernommen werden kann.
- T2: gehört der Klasse *Obsolete (**X**)*, da er im neuen Modell nicht mehr lauffähig ist (da der Übergang von *A4* zu *A2* gelöscht wurde, verbleibt der Zustandsautomat bei der Ausführung mit diesem Testfall im Zustand *A4*).
- T3: gehört zur Klasse *Retest Required (**?**)*, da er lauffähig ist, aber veränderte Modellteile (Zustand *A5*) durchläuft. Dieser Test muss noch einmal durchgeführt werden.

Auf dem geänderten Modell erreicht die Testsuite T also keine 100%ige Transitionsüberdeckung mehr, da die neu eingefügte Transition *t9* nicht überdeckt wird. Deshalb muss ein zusätzlicher Testfall generiert werden, der die neu hinzugekommene Transition überdeckt. Ein Beispiel für solch einen Testfall ist der Testfall T4:

- T4: dieser überdeckt den Pfad *A0, A1, A4, A3, A6*, also auch die neue hinzugekommene Transition *t9*. Somit gehört dieser Testfall zur Klasse *New (**N**)*.

Zur Generierung solcher Testfälle werden die in Abschnitt 4.3 beschriebenen genetischen Algorithmen gezielt eingesetzt. Der Unterschied zur gewöhnlichen Testfallgenerierung besteht darin, dass bei der Generierung der Anfangspopulation[24] die alte Testsuite (für die die Regressionsanalyse durchgeführt wurde) übernommen und im Verlauf der weiteren Generierung nicht

[24]Siehe Abschnitt 4.3.2

mehr verändert wird. Ziel dieser Generierung ist es, die alte Testsuite so um weitere Testfälle zu erweitern, dass sie eine 100%ige Überdeckung auf dem geänderten Modell erreicht, bei gleichzeitiger Minimierung der Anzahl benötigter zusätzlicher Testfälle. Als Beispiel für solch eine Testfallmenge sei TNeu = {T1, T2, T3, T4}, die auf dem geänderten Modell eine 100%ige Transitionsüberdeckung erreicht.

Zur Beschreibung der Funktionsweise der Regressionstestgenerierung, anhand eines konkreten Zustandsautomaten, wurde aus InfotainmentGui[25] eine Testsuite generiert, die 16 Testfälle enthält und eine 100%ige Transitionsüberdeckung erreicht. Danach wurde der Automat wie folgt geändert: zum einen wurde eine Transition gelöscht, dafür wurde eine zusätzliche Transition hinzugefügt und zuletzt wurde der Name eines Zustands geändert. Dies wird in Abbildung 5.10 dargestellt.

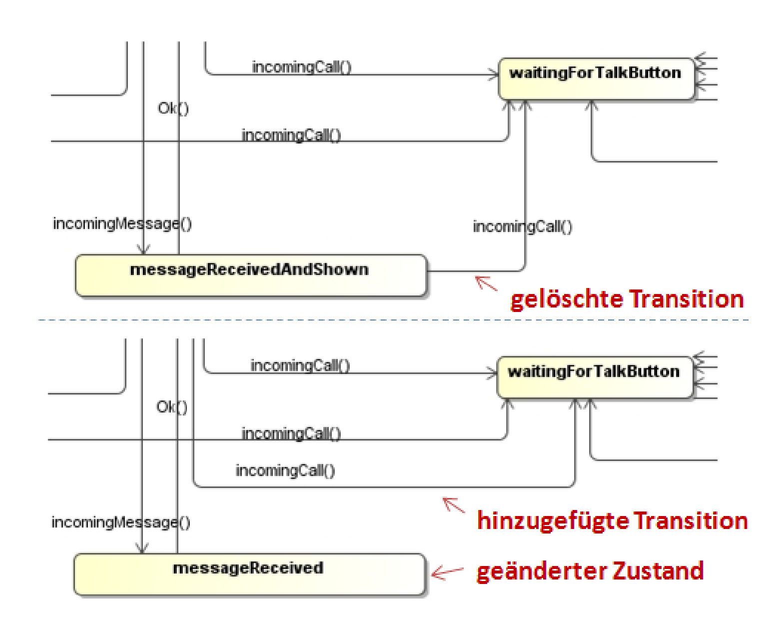


Abbildung 5.10.: Beispiel für Änderung eines Zustandsautomaten: **oben**: Zustandsautomat vor der Änderung; **unten**: Zustandsautomat nach der Änderung

Beim Laden des geänderten InfotainmentGui-Zustandsautomaten erscheint ein Dialog (siehe Abbildung 5.11), da die vorhin beschriebene Testsuite (mit 16 Testfällen) aus der initialen Version des Zustandsautomaten generiert wurde. Der Dialog fragt ab, ob überprüft werden soll, inwiefern die 16 Testfälle auf dem geänderten Zustandsautomaten ausführbar sind. Falls diese Analyse nicht durchgeführt wird, wird die Testsuite verworfen. Falls im Dialog „Yes“ ausgewählt

[25] Diese wird in Abschnitt 6.1 beschrieben

wird, wird die Analyse durchgeführt und die Ergebnisse der durchgeführten Analyse angezeigt, wie in Abbildung 5.12 dargestellt[26]. Da der Testfall „TestCase 3“ den geänderten Zustand durchläuft wird er mit ***?*** markiert. Der Testfall „TestCase 6“ durchläuft auf dem alten Modell die Transition, die im neuen Modell gelöscht wurde, was dazu führt, dass dieser Testfall auf dem geänderten Modell nicht mehr ausführbar ist und mit ***X*** markiert wird. Alle anderen Testfälle können unverändert übernommen werden und werden mit ***I*** markiert.

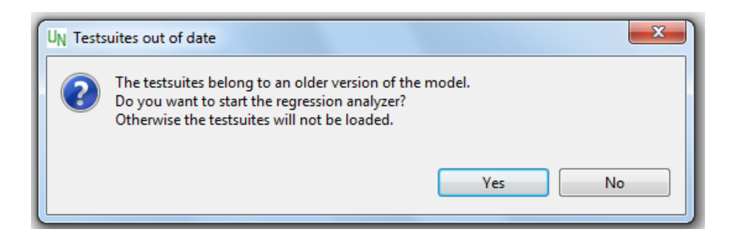


Abbildung 5.11.: Dialog Regressionsanalyse

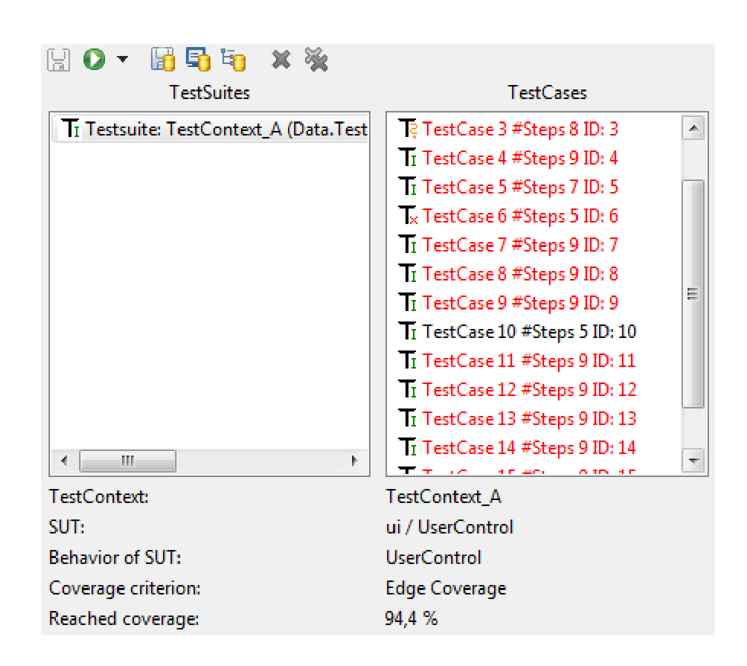


Abbildung 5.12.: Ergebnisse Regressionsanalyse

Wie auch anhand der Abbildung 5.12 zu erkennen ist, erreicht die alte Testsuite auf dem geänderten Zustandsautomat keine 100%ige Überdeckung, da sie nur 94,4% aller Transitionen überdeckt. Um die nicht überdeckten Transitionen zu erreichen, wurde diese Testsuite als Basis für

[26]Die *incomplete Testfälle* werden mit rot markiert. Dies sind Testfälle, die keine kompletten Pfade (die im Anfangszustand starten und im Endzustand enden) durch den Zustandsautomaten durchlaufen

die Generierung einer neuen Testsuite benutzt. Dabei wurden die 16 Testfälle übernommen und im Verlauf des weiteren Generierungsprozesses nicht weiter verändert. Am Ende konnte durch Erweiterung der alten Testsuite um einen zusätzlichen Testfall (Abbildung 5.13: Testfall „TestCase 17“ ist neu dazugekommen und wird mit ***N*** markiert) eine 100%ige Transitionsüberdeckung auf dem geänderten Modell erreicht werden.

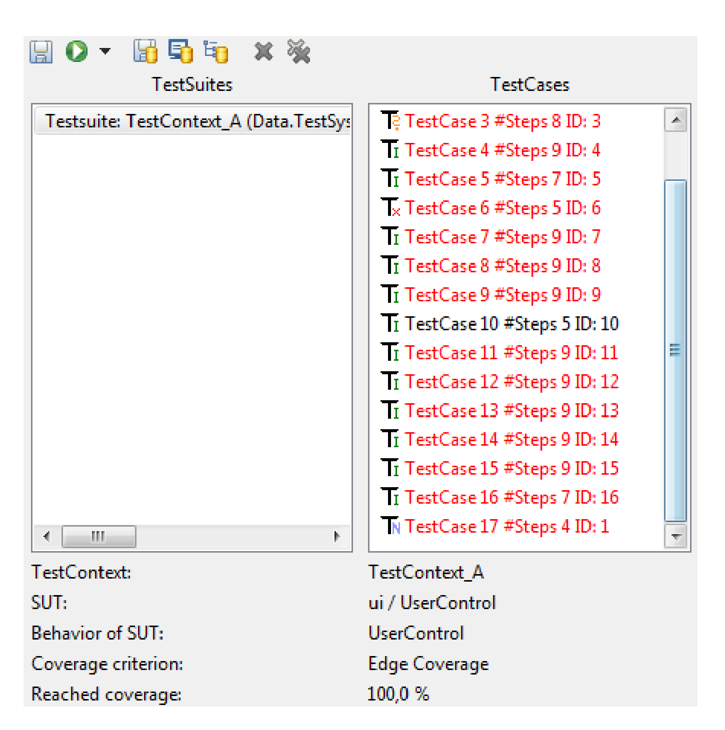


Abbildung 5.13.: Ergebnisse Regressionstestgenerierung

Die Vorgehensweise bei der Regressionstestgenerierung lässt sich dadurch verbessern, dass für die Überdeckung neuer Entitäten nicht komplett neue Testfälle generiert werden, sondern die im neuen Modell nicht mehr lauffähigen Testfälle[27] bei der Generierung mit berücksichtigt werden. Durch Anpassung dieser Testfälle könnten lauffähige Testfälle erhalten werden. So z. B. könnte Testfall T2 (der den Pfad *A0, A1, A4, A2, A3, A6* im Beispielmodell aus Abbildung 5.9 durchläuft) so geändert werden, dass er auch im neuen Modell lauffähig ist (der Pfad im geänderten Modell könnte sein: *A0, A1, A4, A3, A6*).

[27]Testfälle der Klasse *X*

6. Evaluierung der vorgestellten Testfallgenerierungsmethode

Dieses Kapitel beschreibt die Ergebnisse, die bei der automatischen Generierung von Testfällen (mittels des im vorigen Kapitel beschriebenen Werkzeugs) aus Modellen erzielt wurden. Dazu werden zunächst in Abschnitt 6.1 die evaluierten Anwendungen beschrieben, d. h. es werden die Modelle beschrieben, die der Testfallgenerierung zugrunde gelegt wurden. Danach wird in Abschnitt 6.2 beschrieben, welche Parametrisierung für die genetischen Algorithmen bei den durchgeführten Experimenten ausgewählt wurde. Daraufhin werden in Abschnitt 6.3 die bei der Generierung von Komponententestfällen erzielten Ergebnisse beschrieben, gefolgt von der Beschreibung generierter Integrationstestfälle in Abschnitt 6.4. Ein kleiner Teil dieser Ergebnisse wurde bereits in diversen Publikationen veröffentlicht: [OSSP07], [PSO08], [POS08], [SOP08], [SPS09], [SP10].

6.1. Evaluierte Anwendungen

Zur Erprobung des Werkzeugs wurden mehrere Modelle erstellt, die neben der Beschreibung des Softwareverhaltens, auch die vom Benutzer wahrgenommene Mensch-Maschine-Interaktion darstellen. Bei diesen Modellen handelt es sich sowohl um Beispielmodelle als auch um ein Modell, das das Verhalten einer in Entwicklung befindlichen Software beschreibt. Die Diagramme, die das Verhalten der im Folgenden eingeführten Komponenten beschreiben, werden in Anhang A dargestellt.

Liegenprüfplatz

Das aus dem medizintechnischen Umfeld stammende Modell *Liegenprüfplatz* beschreibt das Verhalten einer in Entwicklung befindlicher Software zur Steuerung und Überprüfung von Patienten-

liegen. Überprüft wird dabei, ob die Positionierung der Liege nach bestimmten Liegenbewegungen korrekt ist. Um diese Funktionalität zu realisieren, kontrolliert und verwaltet diese Software mehrere Prüfschritte und erlaubt es am Ende der Durchführung der Schritte ein Protokoll über die ausgeführten Prüfschritte zu speichern. Die Software besteht aus zwei Komponenten: die Komponente *Achstest* ermöglicht es dem Benutzer, die Prüfschritte über eine GUI zu verwalten; die Komponente *PartialResult* ist für das Speichern des Protokolls verantwortlich.

CabinControl

Bei den Beispielmodellen handelt es sich um Modelle, die aus einzelnen oder mehreren kommunizierenden Komponenten bestehen. Zu den Modellen mit mehreren Komponenten zählt das Modell *CabinControl*, das eine Aufzugsteuerung mit sechs Komponenten beschreibt, welche in einem Gebäude mit drei Stockwerken eingesetzt werden kann. Übersichtlich werden die einzelnen Komponenten in Abbildung 6.1 dargestellt; die Pfeile repräsentieren Aufrufe zwischen den Komponenten.

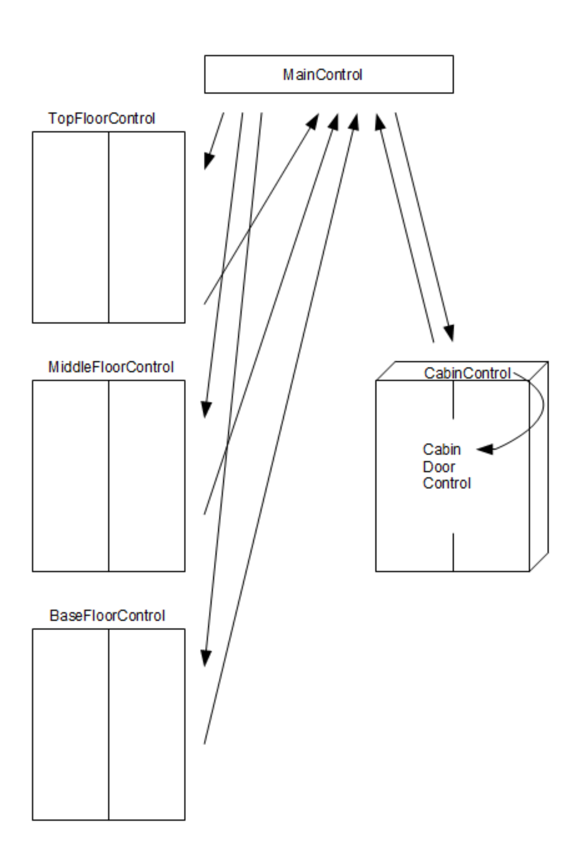


Abbildung 6.1.: Die Komponenten des Modells CabinControl

Pro Stockwerk gibt es eine Stockwerk-Steuerungskomponente (insgesamt gibt es also drei Stockwerk-Steuerungskomponenten: *TopFloorControl*, *MiddleFloorControl* und *BaseFloorControl*), die es erlaubt, den Aufzug im entsprechenden Stockwerk anzufordern. Die zentrale Steuerungskomponente *MainControl* interagiert mit den Stockwerk-Steuerungskomponenten und mit der Kabinensteuerung *CabinControl*. Die Kabinensteuerung interagiert mit der Türsteuerung *CabinDoorControl* der Kabine, um die Türen zu öffnen oder zu schließen.

Infotainment

Des Weiteren beschreibt das Modell *Infotainment* aus dem Kontext der Automobilindustrie ein System, das den Insassen eines Autos eine Reihe von Diensten – z. B. Navigation, Radio und Telefonie – anbietet. Dieses System besteht aus zwei Komponenten. Die zentrale Komponente *InfotainmentGui* stellt die genannten Dienste über eine GUI zur Verfügung. Die zweite Komponente *CDPlayer* ist für das Abspielen von CDs verantwortlich und erhält Befehle von der Komponente InfotainmentGui.

Die Eigenschaften dieser drei Modelle werden in Tabelle 6.1 dargestellt. Die Anzahl der enthaltenen Komponenten wird in der Spalte *AK* angegeben, die Spalte *M* beschreibt die Anzahl der Mappings, die in dem jeweiligen Modell vorhanden sind. Da nicht alle Mappings ausführbar sind[1], gibt die zusätzliche Spalte *M ausf.* die Anzahl der ausführbaren Mappings an.

Modell	AK	M	M ausf.
CabinControl	6	32	26
Infotainment	2	26	19
Liegenprüfplatz	2	73	65

AK:	Anzahl Komponenten
M:	Anzahl Mappings
M ausf.:	Anzahl ausführbare Mappings

Tabelle 6.1.: Kenngrößen der betrachteten UML-Modelle mit mehreren Komponenten

[1] Die Identifikation nicht ausführbarer Mappings wurde manuell durchgeführt

Konto

Neben den Modellen, die mehrere Zustandsautomaten enthalten, wurde ein Beispielmodell erstellt, das nur einen Zustandsautomaten enthält, und zwar das Modell *Konto*. Dieses beschreibt die Zustände eines Geldkontos bei einer Bank. Abhängig von dem Zustand des Kontos (z. B. Guthaben oder Überzogen) kann der Benutzer bestimmte Funktionen abrufen. So kann der Benutzer z. B. – soweit das Konto gedeckt ist – Geld abheben. Dagegen kann Geld jederzeit eingezahlt werden.

Um die Generierung von Komponententestfällen an mehreren Beispielen zu evaluieren, wurden auch Zustandsautomaten aus den in Tabelle 6.1 beschriebenen Modellen einzeln betrachtet: aus Liegenprüfplatz wurde der Zustandsautomat Achstest betrachtet; aus dem Modell Infotainment wurde der Zustandsautomat InfotainmentGui betrachtet.

Die Eigenschaften der drei Zustandsautomaten werden in Tabelle 6.2 beschrieben. Zu jedem Automaten (Spalte *ZM*) werden die Anzahl der enthaltenen Zustände (Spalte *Z*), Transitionen (Spalte *T*) und Transitionspaare (Spalte *TP*) angegeben. Ähnlich wie für Mappings, gilt auch für Transitionspaare, dass manche Paare gar nicht ausführbar sind[2]. Deshalb gibt es eine zusätzliche Spalte, die die Anzahl ausführbarer Transitionspaare angibt (Spalte *TP ausf.*).

6.2. Parametrisierung der Algorithmen

Vor dem Start der Testfallgenerierung müssen die genetischen Algorithmen parametrisiert werden. Dabei stellt sich die Frage, bei welchen Parameterbelegungen der Generierungsprozess am effizientesten funktioniert. Zur Bestimmung von Richtwerten für die einzelnen Parameter wurde das Werkzeug unterschiedlich parametrisiert und ausgeführt bis eine 100%ige *Transitionsüberdeckung* (auf Modellen mit nur einer Komponente) bzw. *Invoking / invoked transitions*-Überdeckung (auf dem Modell CabinControl mit mehreren Komponenten) erreicht wurde. Aus den Parametrisierungen, für die die erzeugte Testfallmenge am kleinsten war, wurde diejenige als Richtwert festgelegt, für die die Erstellung der Testfallmenge am schnellsten war.

Bei der Durchführung dieser Experimente wurde schnell deutlich, dass die optimale Parametri-

[2] Die Identifikation nicht ausführbarer Transitionspaare wurde manuell durchgeführt

ZM	Z	T	TP	TP ausf.
Konto	8	19	64	58
InfotainmentGui	17	54	227	207
Achstest	26	83	342	278

ZM: Zustandsautomat
Z: Anzahl Zustände
T: Anzahl Transitionen
TP: Anzahl Transitionspaare
TP ausf.: Anzahl ausführbare Transitionspaare

Tabelle 6.2.: Kenngrößen der Zustandsautomaten, aus denen Komponententestfälle generiert werden

sierung sehr problemspezifisch ist. Für die betrachteten einfachen Beispiele haben sich Populationsgrößen von 100–250 als optimal erwiesen. Bei der Generierung anhand komplexerer Modelle wurde festgestellt, dass eine Populationsgröße von 250 zu klein ist[3]. Da selbst eine Verdoppelung der Populationsgröße nicht ausreichend war, wurde bei der Generierung ausgehend von dem komplexesten Modell Liegenprüfplatz eine Populationsgröße von 750 ausgewählt. Bei der Generierung von Komponententestfällen haben sich Stagnationsparameter[4] von 2–8 bewährt. Für die Generierung von Integrationstestfällen sollte die lokale Optimierung allerdings abgeschaltet werden, da sie die Entdeckung fehlender Mappings nicht unterstützt.

Aufgrund dieser Erfahrungen wurde bei der Testfallgenerierung für die Modelle Konto, CabinControl und Infotainment die Populationsgröße 250 ausgewählt. Für die Generierung anhand des Liegenprüfplatzmodells (also bei der Generierung von Komponententestfällen aus dem Zustandsautomat Achstest und von Integrationstestfällen anhand des gesamten Modells) wurde die Populationsgröße 750 ausgewählt. Für die Generierung von Komponententestfällen wurde der Stagnationsparameter auf 8 gesetzt.

[3]D. h.: wenn die so parametrisierten genetischen Algorithmen mehrere Male ausgeführt werden, erreichen sie nur in seltenen Fällen eine akzeptable Überdeckung
[4]Siehe 4.3.1

Die der Fitnessfunktion zugewiesenen Gewichte waren 0,99 für Überdeckung und 0,01 für Minimierung der Anzahl der Testfälle. Die Ergebnisse, die bei der Anwendung der auf diese Weise parametrisierten Algorithmen erzielt wurden, werden in den folgenden zwei Abschnitten beschrieben.

6.3. Ergebnisse der Testfallgenerierung für den Komponententest

Um aus einzelnen Zustandsautomaten Testfälle für den Komponententest zu generieren, wurde das in Kapitel 5 beschriebene Werkzeug eingesetzt. Aus jedem der in Tabelle 6.2 beschriebenen Zustandsautomaten wurden drei Testfallmengen generiert, und zwar jeweils eine zur Erfüllung der Kriterien Zustands-, Transitions- bzw. Transitionspaarüberdeckung. Als Abbruchbedingung der Generierung wurde das Erreichen einer 100%igen Überdeckung bezüglich des angestrebten Überdeckungskriteriums ausgewählt. Auf den betrachteten Zustandsautomaten ist jedoch das Kriterium Transitionspaarüberdeckung gar nicht erfüllbar, da jeder Zustandsautomat mehrere nicht ausführbare Transitionspaare enthält[5]. Deshalb wurde in diesen Fällen statt der Überdeckung aller Transitionspaare die Überdeckung aller ausführbaren Transitionspaare angestrebt. Die Ergebnisse der Testfallgenerierung werden in Tabelle 6.3 dargestellt. Ein Teil dieser Ergebnisse wurde bereits in [PS10] vorgestellt.

Mit einer Ausnahme konnten alle generierten Testsuites die 100%ige Überdeckung hinsichtlich der angestrebten ausführbaren Entitäten erreichen. Die Ausnahme bezieht sich auf die Testfallmenge, die 97,84% aller ausführbaren Transitionspaare des Zustandsautomaten Achstest überdeckt.

Tabelle 6.3 zeigt, dass für die Erfüllung der Zustandsüberdeckung nur wenige Testfälle benötigt werden, im Schnitt über die drei Testsuites nur 4,67. Die Anzahl an insgesamt in der Testsuite für Zustandsüberdeckung enthaltenen Teststeps (Spalte TS) ist auch gering, im Schnitt nur 38,7. Zur Erfüllung des Kriteriums Transitionsüberdeckung wurden bedeutend mehr Testfälle benötigt. Die Testsuites für Transitionsüberdeckung enthalten im Schnitt circa viermal so viele Testfälle und Teststeps wie die Testsuites für Zustandsüberdeckung. Einen ähnlichen Unterschied im Umfang gibt es auch zwischen den Testfallmengen für Transitionspaarüberdeckung

[5] Siehe Tabelle 6.2

ZM	Zustandsüb.			Transitionsüb.			Transitionspaarüb.		
	TC	TS	UE	TC	TS	UE	TC	TS	UE
Konto	2	6	8	6	38	19	22	151	58
InfotainmentGui	4	30	17	16	124	54	48	662	207
Achstest	8	80	26	27	284	83	80	1284	272
Mittelwerte	4,67	38,7	17	16,33	148,67	52	50	699	179

ZM: Betrachteter Zustandsautomat
TC: Anzahl generierter Testfälle
TS: Anzahl generierter Teststeps
UE: Anzahl überdeckter Entitäten (Zustände, Transitionen oder Transitionspaare)

Tabelle 6.3.: Ergebnisse der Komponententestgenerierung

und denen für Transitionsüberdeckung.

Erfahrungsgemäß benötigt das Werkzeug UnITeD für die Generierung von Testfallmengen für den Komponententest aus den in Abschnitt 6.1 beschriebenen Modellen meistens nur einige Minuten[6]. Für die Generierung der Testfallmenge, die Transitionspaarüberdeckung auf InfotainmentGui erreicht, wurden allerdings mehrere Stunden benötigt. Allein im Falle der sehr schwierig zu erreichenden Transitionspaarüberdeckung auf dem Zustandsautomaten Achstest konnte selbst nach mehreren Tagen keine 100%ige Überdeckung erreicht werden. In diesem Fall wurde die Generierung manuell durch den Benutzer abgebrochen.

Um zu überprüfen, ob die generierten Testfallmengen weiter optimiert werden können[7], wurde ausgehend von jeder der neun Testfallmengen, die in Tabelle 6.3 beschrieben sind, ein neuer Generierungsvorgang gestartet und nach *200 Generationen* angehalten. Dabei wurde im Rahmen

[6]Die Testfallgenerierung wurde auf einem Rechner mit Intel Core 2 Quad Prozessor (2.66GHz) durchgeführt. Der Rechner verfügt über 4GB Arbeitsspeicher, das darauf installierte Betriebssystem ist Windows 7 (64-Bit-Version)

[7]D. h. es wird überprüft, ob der genetische Algorithmus die Testfallmengen so optimieren kann, dass die erreichte Überdeckung verbessert werden kann (dies kann nur im Falle der Testfallmenge für Transitionspaarüberdeckung auf Achstest erreicht werden, da nur hier die 100%ige Überdeckung nicht erreicht wurde) oder ob die Überdeckung auch mit weniger Testfällen bzw. Teststeps erreicht werden kann

dieses Generierungsschrittes bei der Bewertung der Individuen auch die Anzahl an generierten Teststeps berücksichtigt, mit dem Ziel, die Teststepanzahl auch zu minimieren. Wie dies umgesetzt wurde, wird hier kurz skizziert. Wie schon in Abschnitt 4.3.4 beschrieben, werden vor dem Anwenden der genetischen Operatoren auf eine Population, die Individuen der Population der Fitness nach absteigend sortiert. Die Fitness wird dabei als gewichtete Summe über die Überdeckung und über die Anzahl an Testfällen berechnet, gemäß Formel 4.3. Im Rahmen dieses zusätzlichen Generierungsvorgangs über 200 Generationen wird diese Sortierung dahingehend erweitert, dass bei gleicher Fitness zweier Individuen auch deren Teststepanzahl betrachtet wird. D. h.: wenn zwei Individuen die gleiche Fitness haben, dann wird das Individuum besser bewertet, das weniger Teststeps enthält. Die Ergebnisse des zusätzlichen Generierungsvorgangs werden in Tabelle 6.4 dargestellt. Fett markiert werden Werte, die im Vergleich zu den Werten aus Tabelle 6.3 kleiner sind.

ZM	Zustandsüb.		Transitionsüb.		Transitionspaarüb.	
	TC	TS	TC	TS	TC	TS
Konto	2	6	**4**	**25**	**21**	**133**
InfotainmentGui	4	**21**	**12**	**98**	**40**	**632**
Achstest	8	**76**	**20**	**233**	**73**	**1204**

ZM: Betrachteter Zustandsautomat
TC: Anzahl generierter Testfälle
TS: Anzahl generierter Teststeps

Tabelle 6.4.: Ergebnisse der Testfallgenerierung für den Komponententest nach einem weiteren Optimierungsschritt

Durch diesen zusätzlichen Generierungsvorgang konnten fast alle Testfallmengen minimiert werden. Nicht in allen Fällen konnte die Anzahl an Testfällen verringert werden, dafür konnte aber in den meisten Fällen die Anzahl an Teststeps minimiert werden. Bei den Testfallmengen, die Zustandsüberdeckung erreichen, bestand das geringste Optimierungspotential. Die aus 2 Testfällen bestehende Testfallmenge, die Zustandsüberdeckung auf Konto erreicht, konnte nicht weiter optimiert werden. Bei den anderen zwei Testfallmengen für Zustandsüberdeckung konnten ebenfalls keine Testfälle eingespart werden, dafür konnte aber die Anzahl der Teststeps minimiert werden, zum einen von 30 auf 21 und zum anderen von 80 auf 76.

Im Gegensatz dazu, konnten alle Testfallmengen für Transitionsüberdeckung bedeutend verbessert werden, da sowohl die Anzahl an Testfällen als auch die Anzahl an Teststeps verringert werden konnte. Auch die Testfallmengen für Transitionspaarüberdeckung konnten durch diesen zusätzlichen Generierungsvorgang verbessert werden. So konnte etwa die Transitionspaarüberdeckung für InfotainmentGui mit einer Testsuite bestehend aus 40 Testfällen und 632 Teststeps erreicht werden. Dies bedeutet 8 Testfälle und 30 Teststeps weniger, als die ursprünglich generierte Testfallmenge. Die Testfallmenge für Transitionspaarüberdeckung auf Achstest konnte nicht nur hinsichtlich der Anzahl an Testfällen und Teststeps verbessert werden, sondern auch hinsichtlich der erreichten Überdeckung: mit einer Testsuite bestehend aus 73 Testfällen und 1204 Teststeps (was 7 Testfälle und 80 Teststeps weniger bedeutet, als die ursprünglich generierte Testsuite) werden zusätzlich zwei Transitionspaare überdeckt, was bedeutet, dass die erreichte Überdeckung auf 98,56% steigt.

Die Ergebnisse aus Tabelle 6.4 zeigen, dass es sich besonders im Falle von Testfallmengen für Transitions- oder Transitionspaarüberdeckung lohnen kann, erst einmal den genetischen Generierungsvorgang bis zum Erreichen einer 100%ige Überdeckung laufen zu lassen, um daraufhin eine zusätzliche Optimierungsphase zu starten. Alternativ könnte man nur einen Optimierungsprozess durchführen, der nach einer bestimmten Anzahl an Generationen aufhört. Problematisch dabei ist, dass vor dem Start der Generierung nicht bekannt ist, ob und wenn ja nach wie vielen Generationen die 100%ige Überdeckung erreicht wird. In dieser Hinsicht könnte auch ein weiteres Abbruchkriterium für den Algorithmus sinnvoll sein, wie z. B. „100%ige Überdeckung plus 200 zusätzliche Generationen“.

6.4. Ergebnisse der Testfallgenerierung für den Integrationstest

Neben der Komponententestgenerierung, erlaubt das Werkzeug UnITeD auch die vollautomatische Generierung von Integrationstestfällen. Dabei werden die acht, in Abschnitt 3.3.2 beschriebenen Überdeckungskriterien unterstützt. Bevor die Ergebnisse der Testfallgenerierung beschrieben werden, stellt die Tabelle 6.5 für jedes Kriterium dar, in wie viele Mapping-Gruppen[8] dieses Kriterium die Menge aller Mappings partitioniert. Da Mapping-Gruppen, die nur nicht ausführ-

[8]Siehe Abschnitt 3.3.1

bare Mappings enthalten, nicht überdeckt werden können, stellt die Tabelle 6.5 neben der Anzahl der Mapping-Gruppen, auch in Klammern die Anzahl der ausführbaren Mapping-Gruppen dar (die Klammer wird nur angegeben, falls nicht alle Mapping-Gruppen ausführbar sind).

Generell kann es vorkommen, dass zwei oder mehrere Überdeckungskriterien auf bestimmten Modellen die gleiche Überdeckung fordern, d. h. sie partitionieren die Menge der Mappings gleich. Dies trifft unter anderem für die Kriterien *Pre-states and triggers* und *Pre-states, triggers and effects* bezüglich den Modellen Infotainment und Liegenprüfplatz zu. Zusätzlich dazu fordern – bezogen auf das Modell Infotainment – die Kriterien *Invoking / invoked transitions* und *Invoking transitions on pre-states* die gleiche Überdeckung. Das gleiche trifft im Modell CabinControl auf die Überdeckungskriterien *Triggers and effects in / on pre-states* und *Invoking transitions on pre-states* zu. Auch *Pre-states, triggers and effects* und *Invoking transitions* fordern auf CabinControl die gleiche Überdeckung. In der Tabelle 6.5 wird dies wie folgt dargestellt: falls zwei Kriterien auf einem Modell die gleiche Überdeckung fordern, dann wird nur die rechte Zelle ausgefüllt; in der linken Zelle ist bloß ein Verweis („→“) auf die rechte Nachbarzelle vorhanden.

Modell	K1	K2	K3	K4	K5	K6	K7	K8
CabinControl	13	→	15	14	21(16)	→	25(20)	32(26)
Infotainment	→	13	15	5	18(14)	22(17)	→	26(19)
LPP	→	23	27	21	39(34)	45(40)	69(61)	73(65)

K1: Pre-states and triggers
K2: Pre-states, triggers and effects
K3: Invoking transitions
K4: Effects on pre-states
K5: Effects in / on pre-states
K6: Triggers and effects in / on pre-states
K7: Invoking transitions on pre-states
K8: Invoking / invoked transitions

Tabelle 6.5.: Anzahl der zu überdeckenden Mapping-Gruppen

Die Ergebnisse der Testfallgenerierung für den Integrationstest werden in Tabelle 6.6 beschrie-

ben. Ein Teil dieser Ergebnisse wurde bereits in [PS10] vorgestellt. Jede generierte Testsuite erreicht eine 100%ige Überdeckung bezüglich der ausführbaren Mapping-Gruppen. Für den Fall, dass zwei Kriterien die gleiche Überdeckung fordern, wurden nicht zwei Testfallmengen generiert, die jeweils beide Kriterien erfüllen, sondern nur eine Testfallmenge, die beiden Kriterien genügt. Analog zu Tabelle 6.5, gibt das Symbol „→“ in Tabelle 6.6 an, dass die entsprechenden Testfälle die gleichen sind, wie in den rechten Nachbarzellen.

M	K1		K2		K3		K4		K5		K6		K7		K8	
	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS
C	4	19	→		6	30	3	34	2	33	→		3	41	5	59
I	→		4	36	2	55	2	14	5	60	4	65	→		5	75
L	→		7	96	13	185	12	133	13	171	17	264	33	453	36	504

M:	Betrachtetes Modell
C:	CabinControl
I:	Infotainment
L:	Liegenprüfplatz
TC:	Anzahl generierter Testfälle
TS:	Anzahl generierter Teststeps
K1–K8:	siehe Tabelle 6.5

Tabelle 6.6.: Ergebnisse der Testfallgenerierung für den Integrationstest

Tabelle 6.6 zeigt, dass sich die Hierarchie der Überdeckungskriterien im Allgemeinen auch in der Anzahl an Testfällen und Teststeps widerspiegelt: je anspruchsvoller das Kriterium, desto mehr Testfälle werden für seine Erfüllung benötigt. Allerdings gibt es auch Ausnahmen: z. B. werden für die Erfüllung des Kriteriums *Invoking transitions* auf Infotainment zwei Testfälle generiert, wogegen zur Erfüllung des schwächeren Kriteriums *Pre-states, triggers and effects* vier Testfälle generiert werden. Die Testsuite zur Erfüllung des Kriteriums *Invoking transitions* enthält dafür 19 Teststeps mehr. Das einzige Beispiel für eine Testfallmenge, die sowohl mehr Testfälle als auch mehr Teststeps beinhaltet, als die Testfallmenge, die für ein stärkeres Kriterium generiert wurde, ist die Testfallmenge, die das Kriterium *Effects on pre-states* auf CabinControl erfüllt. Mit 3 Testfällen und 34 Teststeps enthält sie einen Testfall und einen Teststep mehr als die Testfallmenge für *Effects in / on pre-states*. Dieses überraschende Ergebnis lässt sich durch die

angewendete Heuristik erklären, die in diesem konkreten Fall keine befriedigende Minimierung erreicht hat. Die von dem Kriterium *Effects on pre-states* geforderte Überdeckung auf Cabin-Control lässt sich nämlich mit deutlich weniger Testfällen erreichen (mit einem Testfall und 14 Teststeps), was auch in Tabelle 6.7 beschrieben ist.

Experimente zeigten, dass die Dauer der Testfallgenerierung von Integrationstestfällen für die Modelle CabinControl und Infotainment meistens mehrere Minuten beträgt[9]. Die Generierung aus dem weitaus komplexeren Modell Liegenprüfplatz kann selbst im Falle der schwächsten zustandsbasierten Kriterien mehrere Stunden dauern.

Wie auch im Falle der Komponententestfälle wurden die generierten Testfallmengen für den Integrationstest einem weiteren Optimierungsschritt unterzogen, mit dem Abbruchkriterium *200 Generationen*. Die Ergebnisse werden in Tabelle 6.7 beschrieben, wobei fett markiert die Werte dargestellt werden, die geringer sind, als die Werte aus den dazu entsprechenden Zellen der Tabelle 6.6.

M	K1		K2		K3		K4		K5		K6		K7		K8	
	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS	TC	TS
C	**2**	**16**	→		**4**	**22**	**1**	**14**	2	**31**	→		3	**37**	**4**	**49**
I	→		**1**	**32**	2	**50**	**1**	**9**	**1**	**47**	**2**	**51**	→		**3**	**54**
L	→		7	**89**	**11**	**166**	12	**128**	13	**160**	17	**250**	33	**440**	**35**	**477**

M:	Betrachtetes Modell
C:	CabinControl
I:	Infotainment
L:	Liegenprüfplatz
TC:	Anzahl generierter Testfälle
TS:	Anzahl generierter Teststeps
K1–K8:	siehe Tabelle 6.5

Tabelle 6.7.: Ergebnisse der Testfallgenerierung für den Integrationstest nach einem weiteren Optimierungsschritt

[9] Die Testfallgenerierung wurde auf einem Rechner mit Intel Core 2 Quad Prozessor (2.66GHz) durchgeführt. Der Rechner verfügt über 4GB Arbeitsspeicher, das darauf installierte Betriebssystem ist Windows 7 (64-Bit-Version)

Zur übersichtlichen Beschreibung der erreichten Ergebnisse pro Modell werden die Daten aus den vorigen drei Tabellen in jeweils einer Abbildung graphisch aufgearbeitet. Jede Abbildung enthält zwei Diagramme. Im linken Diagramm werden die Anzahl der für das Erreichen einer 100%igen Überdeckung generierten Testfälle pro Kriterium (z. B. in Abbildung 6.2, links: Datenreihe *CabinControl*) dargestellt. Des Weiteren wird für jedes Kriterium die Anzahl der erzeugten ausführbaren Mapping-Gruppen (z. B. in Abbildung 6.2, links: Datenreihe *CabinControl MG.*) beschrieben. Darüber hinaus wird gezeigt, inwiefern die Anzahl an generierten Testfällen durch das Weiterführen des Optimierungsprozesses minimiert werden konnte (z. B. in Abbildung 6.2, links: Datenreihe *CabinControl Min.*). Das rechte Diagramm zeigt die Anzahl der generierten Teststeps vor dem zusätzlichen Optimierungsschritt (z. B. in Abbildung 6.2, rechts: Datenreihe *CabinControl*) und danach (z. B. in Abbildung 6.2, rechts: Datenreihe *CabinControl Min.*).

Bei allen betrachteten Testsuites konnte die Anzahl der Teststeps minimiert werden. Dagegen konnte die Anzahl an benötigten Testfällen nicht immer optimiert werden. Abbildung 6.2 zeigt, dass mit zwei Ausnahmen alle Testsuites, die Integrationstestkriterien auf dem Modell CabinControl erfüllen, hinsichtlich der Testfallanzahl optimiert wurden. Auf dem komplexeren Modell Infotainment konnten mit einer Ausnahme alle Testsuites hinsichtlich Testfallanzahl minimiert werden, wie in Abbildung 6.3 dargestellt. Von den sieben Testfallmengen, die anhand des Modells Liegenprüfplatz generierten wurden, konnten nur zwei hinsichtlich der Testfallanzahl optimiert werden (siehe Abbildung 6.4).

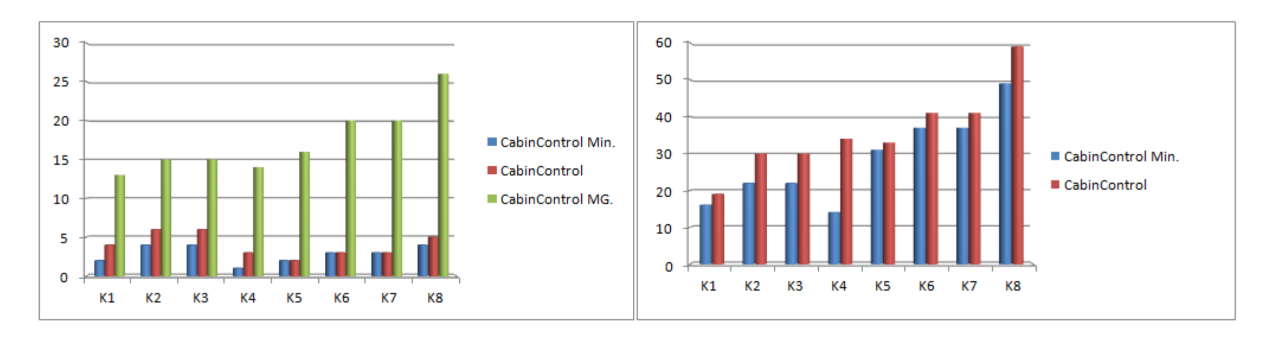


Abbildung 6.2.: Ergebnisse der Integrationstestgenerierung anhand des Modells CabinControl (K1–K8: siehe Tabelle 6.5);
links: Optimierung Testfallanzahl; **rechts**: Optimierung Teststepanzahl

Auch im Falle der Integrationstestfälle macht eine nachfolgende Optimierung also durchaus Sinn. Wie auch im Falle der Komponententestkriterien konnte nicht immer die Anzahl der Testfälle minimiert werden, dafür aber die Anzahl der Teststeps.

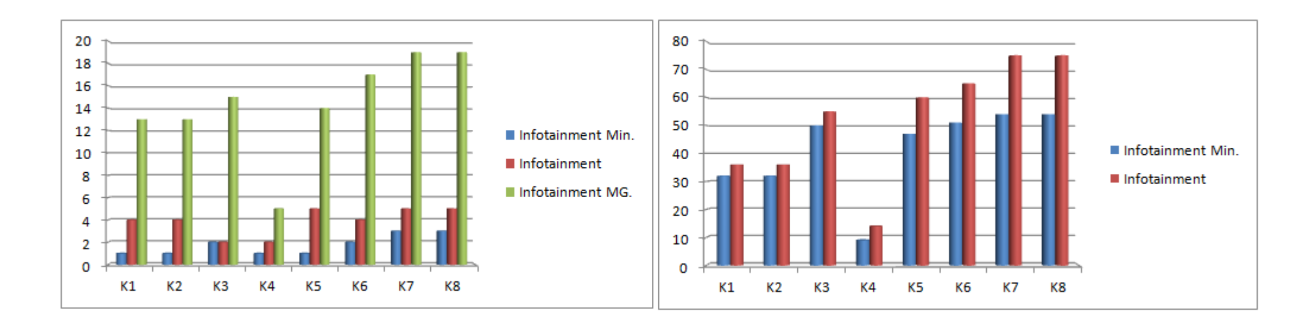


Abbildung 6.3.: Ergebnisse der Integrationstestgenerierung anhand des Modells Infotainment (K1–K8: siehe Tabelle 6.5);
links: Optimierung Testfallanzahl; **rechts**: Optimierung Teststepanzahl

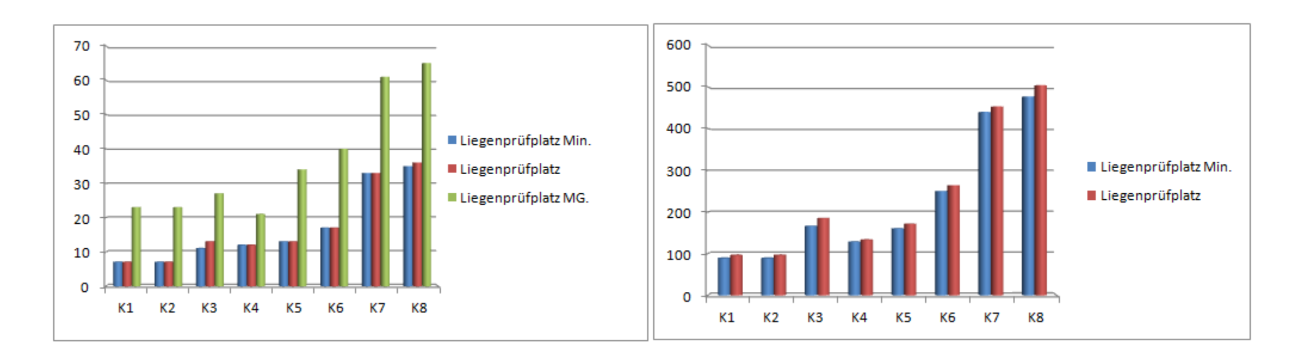


Abbildung 6.4.: Ergebnisse der Integrationstestgenerierung anhand des Modells Liegenprüfplatz (K1–K8: siehe Tabelle 6.5);
links: Optimierung Testfallanzahl; **rechts**: Optimierung Teststepanzahl

7. Evaluierung des Fehlererkennungspotentials modellbasierter Testfälle

Nachdem im vorigen Kapitel die Ergebnisse der automatischen, modellbasierten Testfallgenerierung vorgestellt wurden, wird in diesem Kapitel das *Fehlererkennungspotential modellbasierter Testfälle* evaluiert, indem auf die Anzahl und Art der erkannten Fehler eingegangen wird. Im Folgenden ist auch vom *Fehlererkennungspotential eines Überdeckungskriteriums* die Rede; damit ist immer das Fehlererkennungspotential automatisch generierter Testfallmengen gemeint, die zur Erfüllung dieses Kriteriums generiert wurden. Wie schon im letzten Kapitel erwähnt, erfüllen – mit einer Ausnahme – alle generierten Testfallmengen das angestrebte Überdeckungskriterium zu 100%. Die Ausnahme bezieht sich auf eine Testfallmenge, die nicht alle Transitionspaare des Zustandsautomaten Achstest überdeckt[1]. Da mit Hilfe modellbasierter Testfälle *Modellierungsfehler* oder *Implementierungsfehler* erkannt werden können, wird im weiteren Verlauf das Erkennungspotential hinsichtlich dieser zwei Fehlerkategorien in den Abschnitten 7.1 und 7.2 separat behandelt. Ein Teil dieser Ergebnisse wurden bereits in [PS10] veröffentlicht.

7.1. Erkennungspotential hinsichtlich Modellierungsfehler

Zur Bewertung des Potentials modellbasierter Testfälle hinsichtlich der Erkennung von Modellierungsfehlern wurde ein *Modell-Mutationstest* durchgeführt. Um dieses Konzept zu erklären, werden zunächst im folgenden Unterabschnitt die Grundlagen des Mutationstests beschrieben.

[1] Siehe Abschnitt 6.3

7.1.1. Mutationstest

Der *Mutationstest* ist eine Testart, die sich von den in Abschnitt 2.5.2 beschriebenen Testarten grundlegend unterscheidet: hierbei geht es nicht mehr darum, Softwarefehler zu finden, sondern um die Bewertung einer Testsuite hinsichtlich ihres Fehlererkennungspotentials.

Der Mutationstest basiert auf zwei wichtigen Annahmen. Zum einen auf der Annahme, dass erfahrene Programmierer fast korrekte Programme schreiben, die nur geringfügig vom vorgesehenen Verhalten abweichen (Engl. *Competent Programmer Hypothesis*). Zum anderen wird angenommen, dass komplexe Fehler an einfache Fehler gekoppelt sind. Das heißt, dass Testfälle, die in der Lage sind, einfache Fehler zu entdecken, sich auch dazu eignen, komplexere Fehler zu finden (Engl. *Coupling Effect Hypothesis*).

Ansätze zur Bewertung von Testsuites mittels Mutationstest sind vor allem auf Codeebene sehr verbreitet. So z. B. werden in [Ost07], [DMM01], [WS08], [ABLN06] alle Testfälle einer Testsuite auf der Originalversion und auf geänderte Versionen der Software (so genannte *Mutanten*, welche durch so genannte *Mutationsoperatoren* entstehen) ausgeführt. Durch Vergleich des sich dabei ergebenden Verhaltens wird bewertet, ob ein Testfall bestimmte Mutanten zu entdecken erlaubt. Eine Testsuite erkennt einen Mutanten, falls mindestens ein darin enthaltener Testfall den Mutanten erkennt.

Abhängig von der Art in der überprüft wird, ob ein Testfall einen Mutanten erkennt, können Mutationstestverfahren wie folgt klassifiziert werden [Lig09]:

- *Starker Mutationstest* (Engl. *Strong Mutation Test*): im Rahmen eines starken Mutationstests wird die Bewertung der Testfälle hinsichtlich der Erkennbarkeit eines Mutanten nur anhand abweichender Ausgaben des Mutanten gegenüber dem originalen Programm durchgeführt.
- *Schwacher Mutationstest* (Engl. *Weak Mutation Test*): Im Gegensatz zum starken Mutationstest wird der Mutant als erkannt markiert, falls die eingefügte Programmmodifikation zur Folge hat, dass sich bei Ausführung des Testfalls die Zwischenzustände (z. B. die Variablenbelegungen) des Programms und des Mutanten unterscheiden.

Während es beim schwachen Mutationstest für die Erkennung eines Mutanten ausreicht, dass sich die Zwischenzustände im Mutanten vom initialen Programm unterscheiden, muss beim starken Mutationstest der unterschiedliche Zwischenzustand eine unterschiedliche Ausgabe zur Fol-

ge haben[2].

Typische Mutationsoperatoren auf Ebene des Programmcodes sind laut [Lig09]:

- Referenz einer anderen Variablen
- Definition einer anderen Variablen
- Verfälschung arithmetischer Ausdrücke um multiplikative oder additive Konstanten oder Veränderung von Koeffizienten
- Verfälschung boolescher Ausdrücke

Durch Anwendung einer einzigen Transformation auf den Programmcode entsteht ein so genannter *Mutant erster Ordnung* (Engl. *First Order Mutant*). Im Gegensatz dazu, wird durch Anwendung mehrerer Operatoren auf das gleiche Programm ein *Mutant höherer Ordnung* (Engl. *Higher Order Mutant*) erzeugt [JH09]. Die meisten Ansätze zum Mutationstest betrachten nur Mutanten erster Ordnung; so auch der im Folgenden beschriebene Modell-Mutationstestansatz.

7.1.2. Modell-Mutationstest

Ansätze zum *Modell-Mutationstest* übertragen die Ideen des Mutationstests von der Codeebene auf die Ebene der Modelle. Somit soll das Potential von modellbasiert generierten Testfällen hinsichtlich der Erkennung von Modellierungsfehlern bewertet werden. So z. B. werden in [Cho78] *Endliche Zustandsmaschinen* (Engl. *Finite State Machine*, Kurz *FSM*) betrachtet. Dabei wurden typische Fehler bei der Modellierung von FSM vorgestellt und klassifiziert. Danach wurde analysiert, inwiefern Testfälle, die eine strukturelle Überdeckung der FSM erreichen, solche Fehler zu entdecken erlauben. Der Ansatz [FMSM99] baut auf der Arbeit [Cho78] auf und betrachtet *Statecharts*. Im Vergleich zu *FSM* stellen Statecharts zusätzliche Sprachelemente (z. B. History-Zustände oder Sprachelemente, die es erlauben, hierarchische und parallele Komposition von Zuständen zu beschreiben) zur Verfügung. Die in [FMSM99] beschriebenen Fehlerarten betreffen deshalb auch Fehler, die bei der Modellierung solcher zusätzlichen Konstrukte auftreten können. In [SMFS00] werden Mutationsoperatoren für die Sprache *Extended Finite State Machine Language* (Kurz *Estelle*) vorgestellt. Diese Sprache erlaubt es Systeme zu spezifizieren, die aus mehreren kommunizierenden Modulen bestehen (Kommunikation erfolgt über so genannte *Bi-directional Channels*). *Erweiterte Endliche Zustandsmaschinen* (Engl. *Extended Finite State*

[2] Siehe Zusammenhang fehlerhafter Zustand und Programmversagen in Abschnitt 2.4

Machines, Kurz *EFSM*) beschreiben dabei das Verhalten der einzelnen Module.

Von den gerade betrachteten Ansätzen ist der hier beschriebene Modell-Mutationstestansatz am ehesten mit der Arbeit von [SMFS00] vergleichbar, allerdings sind aufgrund der Unterschiede in der benutzten Modellierungssprache die betrachteten Fehlerarten unterschiedlich. Während [SMFS00] *EFSMs* betrachtet, werden für die Modellierung des Verhaltens einzelner Module im Rahmen dieser Arbeit *UML-Zustandsautomaten* benutzt. Des Weiteren wird die Kommunikation unterschiedlich dargestellt: durch *Bi-directional Channels* in [SMFS00] und durch *Aufrufe in Effects von Transitionen* in dieser Arbeit.

Die Durchführung eines Mutationstests ist meistens so aufwendig, dass sie ohne Werkzeugunterstützung nicht zu bewerkstelligen ist. Um die Evaluierung des Fehlererkennungspotentials modellbasierter Testfälle so weit wie möglich zu automatisieren, wurde das in Kapitel 5 beschriebene Werkzeug UnITeD um ein Modul erweitert, das diese Bewertung unterstützt, indem es zwei Funktionalitäten zur Verfügung stellt:

- *Mutantengenerierung*: In einem ersten Schritt werden zunächst Mutanten durch Einfügen von Änderungen in ein so genanntes *initiales Modell*[3] erzeugt. Dazu wurden eine Reihe von Modell-Mutationsoperatoren implementiert; die Details zu den einzelnen Operatoren werden in Abschnitt 7.1.2.2 beschrieben.
- *Mutationsevaluation*: Im Anschluss an die Generierung von Mutanten werden diese mit der zu bewertenden Testsuite ausgeführt, um zu bestimmen, welche Mutanten durch die Testsuite erkannt werden. Daraufhin wird der so genannte *Mutation-score* (kurz *MS*) berechnet, der den prozentualen Anteil an erkannten Mutanten wiedergibt. Neben der Berechnung des Mutation-scores erlaubt diese Evaluation auch die Ergebnisse bezüglich einzelner Klassen von Mutationen zu bestimmen[4]. Die Formel für die Berechnung des Mutation-scores einer Testsuite TS_i, die aus einem Modell MO generiert wurde, lautet:

$$MS_{TS_i} = \frac{|D(MO,\, TS_i)|}{|M(MO)| - |E(MO)|} \tag{7.1}$$

Dabei ist $M(MO)$ die Menge der anhand des Modells MO generierten Modellmutanten, $E(MO)$ die Menge der Modellmutanten, die funktionsäquivalent zum originalen Modell

[3] Ist das Modell, aus dem die zu bewertenden modellbasierten Testfälle erzeugt wurden

[4] Siehe Unterabschnitt 7.1.3.1 und 7.1.4.1

sind und $D(MO, TS_i)$ die Menge der Mutanten, die durch die Testsuite TS_i erkannt wurden.

Eine modellbasierte Testsuite erkennt einen Mutanten (*tötet* einen Mutanten), wenn mindestens ein darin enthaltener Testfall den Mutanten tötet, d. h. wenn sich das *nach außen sichtbare Verhalten* bei der Ausführung des Testfalls auf dem Mutanten von dem nach außen sichtbaren Verhalten bei der Ausführung des Testfalls auf dem initialen Modell unterscheidet.

Zur Erklärung des Konzepts *nach außen sichtbares Verhalten* muss zunächst das Konzept der *externen Zustände* eingeführt werden. Diese Zustände beschreiben im Allgemeinen vom Benutzer wahrnehmbares Systemverhalten z. B. in Form von Dialogen. Welche Zustände als externe Zustände zu verstehen sind, muss bei der Modellierung festgelegt werden. Dazu muss an den entsprechenden Zuständen der Stereotyp *«ExternalSt»* angebracht werden. Eine Folge von externen Zuständen wird als *externer Pfad* bezeichnet. Um dieses Konzept beispielhaft darzustellen, werden die Zustandsautomaten aus Abbildung 7.1 zusammen mit den in Abschnitt zur Modellsimulation beschriebenen Testfällen (ein Testfall für den Komponententest und einer für den Integrationstest) betrachtet.

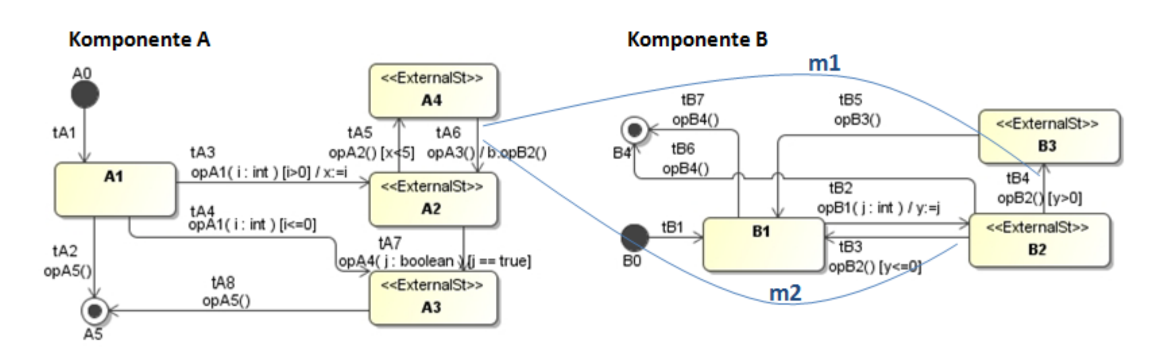


Abbildung 7.1.: Beispiel für ein Modell mit zwei interagierenden Komponenten, die externe Zustände enthalten

Als Beispiel für ein Komponententestfall wurde in Abschnitt 4.3.3.1 folgender Testfall für Komponente *A* beschrieben:

opA1(88742965), opA4(true), opA5().

Der Pfad, der bei der Abarbeitung des Testfalls durch den Zustandsautomaten von *A* durchlaufen wird, ist folgender:

Pfad durch *A*: *A0, A1, A2, A3, A5*.

Für die Mutationsevaluation ist allerdings nicht der ganze Pfad, sondern nur der Pfad bestehend aus externen Zuständen relevant:

externer Pfad durch *A*: *A2, A3*.

Der externe Pfad stellt das nach außen sichtbare Verhalten bei der Ausführung eines Komponententestfalls dar. Wenn sich also bei der Ausführung des Komponententestfalls auf dem Mutant die Folge der durchlaufenen externen Zustände von der Folge der durchlaufenen externen Zustände auf dem initialen Modell unterscheidet, wird der Mutant als erkannt markiert.

Im Falle der Evaluierung von Integrationstestfällen ist die Frage nach dem nach außen sichtbaren Verhalten etwas komplizierter, da hier Pfade durch alle interagierenden Komponenten betrachtet werden müssen. Zur Illustration dieses Konzepts sei der Integrationstestfall:

opA1(4), opB1(69171531), opA2(), opA3().

Bei Abarbeitung dieses Testfalls werden folgende Pfade durch die Zustandsautomaten aus Abbildung 7.1 beschrieben:

Pfad durch *A*: *A0, A1, A2, A4, A2*
Pfad durch *B*: *B0, B1, B2, B3*.

Durch Fokussierung auf die externen Zustände ergeben sich folgende externe Pfade:

externer Pfad durch *A*: *A2, A4, A2*
externer Pfad durch *B*: *B2, B3*.

Das nach außen sichtbare Verhalten bei der Ausführung eines Integrationstestfalls ist also eine Menge von externen Pfaden. Bei der Überprüfung, ob ein Integrationstestfall einen bestimmten Mutanten erkennt, werden pro Komponente die externen Pfade auf dem initialen Modell und auf dem Mutanten betrachtet. Falls für mindestens eine Komponente gilt, dass sich der externe Pfad auf dem initialen Modell und auf dem Mutanten unterscheidet, dann wird der Mutant als erkannt markiert.

Beispiel Anschaulich wird das Konzept der externen Zustände vor allem bei Betrachtung des Modells CabinControl, das in Abschnitt 6.1 beschrieben wurde. Hier interagieren die Stockwerk-Steuerungskomponenten mit der MainControl, die wiederrum mit der Kabinensteuerung CabinControl interagiert. Die Komponente MainControl läuft also im Hintergrund und ist für die Koordination der Stockwerk-Steuerungskomponenten mit der Kabinensteuerung verantwort-

lich. Das bedeutet, dass der Zustand, in dem sich MainControl befindet, für den Benutzer dieses Systems nicht erkennbar ist[5]. Der Benutzer kann nur beurteilen, ob z. B. die Türen auf einem Stockwerk offen oder geschlossen sind.

Die gerade beschriebe Vorgehensweise entspricht dem in der Literatur zum Code-basierten Mutationstest bekannten starken Mutationstest, da bei der Bewertung des Testfalls hinsichtlich seiner Fähigkeit den Mutanten zu erkennen nur die Ausgaben des Systems betrachtet werden. Im Gegensatz zu diesem Vorgehen hätte ein schwacher Modell-Mutationstestansatz bedeutet, dass die Analyse alle Zustände betrachtet, die bei der Abarbeitung des Testfalls durchlaufen wurden.

Ein Mutant kann nicht erkannt werden, wenn er funktionsäquivalent zum initialen Modell ist. In diesem Fall gibt es keinen Testfall, der diesen Mutanten erkennt. Die in Abschnitt 7.1.3 und 7.1.4 beschrieben Mutation-scores wurden alle ohne Berücksichtigung der funktionsäquivalenten Mutanten berechnet. Somit ist der Wert für $|E(MO)|$ immer gleich 0. Deshalb sind die im Folgenden angegebenen Mutation-scores auch als *pessimistisch* zu bezeichnen.

Bevor sinnvolle Modell-Mutationsoperatoren definiert werden können, müssen zunächst typische Modellierungsfehler für Zustandsautomaten identifiziert und klassifiziert werden. Diese Klassifikation wird im folgenden Unterabschnitt beschrieben, gefolgt von der Beschreibung der umgesetzten Modell-Mutationsoperatoren.

7.1.2.1. Fehlerarten bei der Modellierung kommunizierender UML-Zustandsautomaten

Bei der Modellierung des Verhaltens kommunizierender Komponenten mittels Zustandsautomaten können Fehler an den Transitionen oder an den Zuständen auftreten. Die Tatsache, dass eine Transition aus Trigger, Guard, Effect, Pre-state und Post-state zusammengesetzt ist und dass ein Effect sowohl Variablenzuweisungen als auch Aufrufe anderer Komponenten beinhalten kann, wird bei der unten beschriebenen Klassifikation berücksichtigt. Wie schon in Abschnitt 3.1 erwähnt, wurden bei der Modellierung nur eine Teilmenge der für Zustandsautomaten zulässigen Konstrukte benutzt. Daher werden hier nur Fehler beschrieben, die sich auf die eingesetzten Konstrukte beziehen. Diese Fehler können wie folgt klassifiziert werden:

- Trigger-Fehler

[5] Deshalb ist an keinem der Zustände von MainControl der Stereotyp «ExternalSt» angebracht. Der Zustandsautomat der Komponente MainControl wird in Anhang A beschrieben

  - fehlende Trigger
  - zusätzliche Trigger
  - inkorrekte Trigger
- Guard-Fehler
  - fehlende Guards
  - zusätzliche Guards
  - inkorrekte Vergleichsoperatoren in Guards (z. B. $<=$ statt $<$)
  - inkorrekte boolesche Operatoren in Guards (z. B. && statt $||$)
  - Bezug auf inkorrekte Variablen
- Effect-Fehler
  - Variablenzuweisung-Fehler
    - fehlende Zuweisungen
    - zusätzliche Zuweisungen
    - inkorrekte arithmetische Operatoren (z. B. $+$ statt $*$)
    - Bezug auf inkorrekte Variablen
  - Aufruf-Fehler
    - fehlende Aufrufe
    - zusätzliche Aufrufe
    - inkorrekte Aufrufe
  - Generische Effect-Fehler
    - fehlende generische Effects
    - zusätzliche generische Effects
- Pre-state-Fehler
  - inkorrekter Pre-state einer Transition
- Post-state-Fehler
  - inkorrekter Post-state einer Transition

- Transition-Fehler
  - fehlende Transitionen
  - zusätzliche Transitionen
- Zustand-Fehler
  - fehlende Zustände
  - zusätzliche Zustände

Bei Betrachtung der Auswirkungen der oben beschriebenen Fehler wird klar, dass manche Fehler schon während der Komponententestphase aufgedeckt werden könnten. Im Gegensatz dazu, können sich andere Fehler nur beim Zusammenspiel mehrerer Komponenten manifestieren. Dementsprechend können die gerade beschriebenen Fehler auch folgenden Kategorien zugeordnet werden.

- *intramodulare Fehler*: Fehler, die beim Test einzelner Komponenten aufgedeckt werden können.
- *intermodulare Fehler*: Fehler, die sich nur äußern, wenn die fehlerhafte Komponente mit anderen Komponenten interagiert. Beispiele für solche Fehler sind Aufruf-Fehler, die nur durch einen Integrationstest entdeckt werden können.

7.1.2.2. Modell-Mutationsoperatoren

Zur automatischen Generierung von Modellmutanten wurden Modell-Mutationsoperatoren (werden im Folgenden auch einfach Operatoren[6] genannt) implementiert, die nacheinander auf einem initialen Modell ausgeführt werden und somit eine Menge von Mutanten erzeugen. Alle Operatoren gehen nach dem in Abbildung 7.2 beschriebenen generischen Muster vor. Mit Ausnahme des Operators Zustand-Löschen (der alle Zustände betrachtet) betrachten die einzelnen Operatoren ausgewählte Transitionen aus den Zustandsautomaten des Modells.

In Abbildung 7.2 beschreibt $betrachteteEntitäten$ die Menge der betrachteten Transitionen oder Zustände. Diese Menge enthält alle Transitionen eines Zustandsautomaten, falls die Operatoren auf ein Modell angewendet werden, das nur eine Komponente C mit einem dazugehörigen Zustandsautomaten enthält ($betrachteteEntitäten = T_C$). Analog beinhaltet die Menge aller be-

[6]Nicht zu verwechseln mit den genetischen Operatoren, die in Abschnitt 4.3.4 beschrieben werden

```
Parameter:
Modell: das initiale Modell
betrachteteEntitäten: Menge an betrachtenden Transitionen/Zuständen
aus dem Modell

for all t in betrachteteEntitäten do
      if bedingungErfüllt(t) then
            mutantenMenge = operator(t);
            for all m in mutantenMenge do
                  if mutantSyntaktischKorrekt(m) then
                        speichere(m)
                   endif
            endfor
      endif
endfor

Ergebnis:
Menge von Mutanten, die auf der Festplatte abgespeichert werden
```

Abbildung 7.2.: Pseudocode zur generischen Beschreibung der Vorgehensweise einzelner Modell-Mutationsoperatoren

trachteten Zustände alle Zustände des Zustandsautomaten ($betrachteteEntitäten = S_C$). Durch Anwendung der Operatoren auf einen einzelnen Zustandsautomaten sollen also intramodulare Fehler simuliert werden. Bei Anwendung der Operatoren auf einem Modell, das mehrere kommunizierende Zustandsautomaten enthält sollen intermodulare Fehler eingefügt werden. In diesem Fall betrachten die Operatoren nicht mehr alle Transitionen der Zustandsautomaten, sondern nur die Transitionen, die zu Mappings gehören. Die Zustand-Operatoren werden in diesem Fall nicht angewendet.

Zur Beschreibung der Operatoren werden hier nur die Operatoren betrachtet, die für Transitionen definiert sind. Die Definition der Operatoren auf Zustände erfolgt analog. Diese gehen wie folgt vor: erstens wird für jede Transition überprüft, ob die Bedingung[7] für das Erzeugen eines Mutanten anhand dieser Transition erfüllt ist (in Abbildung 7.2: *bedingungErfüllt(t)*). Falls dies nicht der Fall ist, wird die Transition nicht weiter betrachtet und es wird zur nächsten übergegangen. Falls die Bedingung erfüllt ist, wird der Operator eine Menge von Mutanten erzeugen. Die

[7] Diese Bedingung ist operatorenspezifisch; pro Operator wird die dazugehörige Bedingung bei der Beschreibung des Operators angegeben

meisten Modell-Mutationsoperatoren erzeugen pro Transition nur einen Mutant, wenige Operatoren (z. B. der Guard-Variable-Ändern-Operator) können aber mehrere Mutanten erzeugen[8].

Alle erzeugten Mutanten werden in einem weiteren Schritt überprüft (in Abbildung 7.2: *mutantSyntaktischKorrekt(m)*). Falls der betrachtete Mutant syntaktisch korrekt ist, wird er auf der Festplatte abgespeichert (in Abbildung 7.2: *speichere(m)*), falls nicht, wird er verworfen. Weshalb diese zusätzliche Überprüfung erforderlich ist, wird anhand des folgenden Beispiels erklärt. Wird der Operator Trigger-Ändern auf der Transition *tA3* aus Abbildung 3.1 angewendet, tauscht er den Trigger dieser Transition durch einen Trigger einer anderen Transition aus[9], z. B. durch den Trigger der Transition *tA2*. Dann ändert sich die Transition *tA3* wie folgt:

$$\textbf{aus}\quad tA3 = (A1,\ opA1(i:int),\ i>0,\ x:=i,\ A2)$$
$$\textbf{wird}\ tA3 = (A1,\ opA5()\qquad,\ i>0,\ x:=i,\ A2).$$

Durch diese Änderung ist der Guard und der Effect der Transition *tA3* ungültig geworden, da die Variable *i* (die im Guard und im Effect referenziert wird) nicht mehr definiert ist.

Des Weiteren kann durch Änderung des Triggers einer Transition ein nichtdeterministischer Zustandsautomat erzeugt werden. Dies kommt dann vor, wenn nach der Änderung des Triggers im Automaten zwei Transitionen den gleichen Pre-state und den gleichen Trigger (bei gleichzeitig nicht disjunkten Guards) haben. Da durch das Werkzeug solche nichtdeterministischen Mutanten nicht automatisch erkannt werden können, muss bei der Generierung der Mutanten darauf geachtet werden, dass möglichst nur deterministische Mutanten erzeugt werden. Dazu wurden bei der Umsetzung der einzelnen Operatoren sowohl die Bedingung für die Durchführung einer Mutation an einer Transition oder einem Zustand als auch die Operation an sich entsprechend umgesetzt. Wie dies pro Operator realisiert wurde, wird in folgender Form beschrieben:

Mutationsbedingung (Kurz B): für jeden Modell-Mutationsoperator gibt B die Bedingung an, die die Entität (Transition oder Zustand) erfüllen muss, damit dieser Operator auf dieser Entität angewendet wird

Mutationsoperation (Kurz O): beschreibt die Operation, die durchgeführt wird

Zur kompakten Beschreibung der Bedingung (B) und der durch die Operatoren durchgeführten

[8]Für Details zur genauen Funktionsweise des Operators: siehe Beschreibung von Guard-Variable-Ändern
[9]Für Details zur genauen Funktionsweise des Operators: siehe Beschreibung von Trigger-Ändern

Änderung (O) werden die in Abschnitt 3.1 beschriebenen Mengen (T_C, S_C, Tr_C usw.) benutzt. Darüber hinaus werden noch ein paar zusätzliche Begriffe benötigt, die hier eingeführt werden. Für eine Komponente $C \in COMP$ und eine Transition $t \in T_C$ sei:

$MengeKandidatenTrigger_t := Tr_C \setminus \{tr_1 \in Tr_C | \exists t_1 \in T[Tr]_C(tr_1) : pre(t_1) = pre(t)\}$: für eine bestimmte Transition *t* beschreibt diese Menge die Menge der Trigger der Zustandsmaschine, die als Ersatz für den Trigger dieser Transition bei einer Mutation in Frage kommen. Um Nichtdeterminismus zu vermeiden, kommen hier nur die Trigger in Frage, die nicht als Trigger für andere Transitionen auftreten, die aus dem Pre-state der Transition *t* ausgehen.

$MengeKandidatenPreState_t := S_C \setminus S[Tr]_C(tr(t))$: für eine bestimmte Transition *t* beschreibt diese Menge, die Menge der Zustände des Zustandsautomaten, die als Ersatz für den Pre-state dieser Transition in Frage kommen. Um Nichtdeterminismus zu vermeiden, kommen hier nur die Pre-states in Frage, die keine ausgehende Transition haben, deren Trigger gleich dem Trigger der Transition *t* ist.

$MengeTransitionenFürGuard_C := \{t \in T_C | tr(t) = [_]\} \cup \{t \in T_C | \nexists t_1 \in T_C : t \neq t_1 \wedge pre(t) = pre(t_1) \wedge tr(t) = tr(t_1)\}$: diese Menge beinhaltet Transitionen, auf die *Guard-Operatoren* angewendet werden sollen. Da durch Änderungen an den Guards nichtdeterministische Automaten erzeugt werden können, beinhaltet diese Menge zum einen alle Transitionen ohne Trigger und zum anderen die Transitionen mit Trigger, für die folgendes gilt: es existiert im Zustandsautomat keine andere Transition mit dem gleichen Trigger und dem gleichen Pre-state, wie die aktuell betrachtete Transition.

Des Weiteren werden noch folgende Mengen eingeführt:

$MengeVergleichsoperatoren := \{<, <=, >, >=\}$: diese Menge beinhaltet einen Teil der bei der Mutation betrachteten Vergleichsoperatoren.

$MengeArithmetischeOperatoren := \{+, *, /, -\}$: diese Menge beinhaltet alle bei der Mutation betrachteten arithmetischen Operatoren.

Aufbauend auf diesen Konzepten werden im Folgenden die implementierten Modell-Mutationsoperatoren beschrieben. Dabei lassen sich die Operatoren in sieben Klassen aufteilen: Trigger-Operatoren, Guard-Operatoren, Effect-Operatoren, Pre-state-Operatoren, Post-state-Operatoren, Transition-Operatoren und Zustand-Operatoren. Die Klasse Effect-Operatoren lässt sich weiter auf folgende drei Klassen aufteilen: Variablenzuweisung-Operatoren, Aufruf-Operatoren und Generische-Effect-Operatoren.

Trigger-Operatoren

Trigger-Löschen: dieser Operator löscht den Trigger einer Transition t. Dadurch kann es passieren, dass der Guard und/oder der Effect der Transition ungültig werden[10]. Deshalb wird dieser Operator nur auf Transitionen angewendet, die zwar einen Trigger aber weder Guard noch Effect besitzen. Darüber hinaus muss gelten, dass der Pre-state der Transition t nur eine einzige ausgehende Transition hat, und zwar die Transition t.

B: $tr(t) \neq [_] \wedge g(t) = [_] \wedge e(t) = [_] \wedge |Ak_{pre(t)}| = 1$.

O: $tr(t)$ wird gelöscht.

Trigger-Hinzufügen

B: $tr(t) = [_] \wedge pre(t) \neq Anfangszustand_C$.

O: der Transition t wird ein Trigger zugewiesen, der zufällig aus der Menge $Tr_C \setminus \{[_]\}$ ausgewählt wird.

Trigger-Ändern

B: $tr(t) \neq [_]$.

O: $tr(t)$ wird durch einen zufällig aus der $MengeKandidatenTrigger_t \setminus \{[_]\}$ ausgewählten Trigger ersetzt.

Guard-Operatoren

Guard-Löschen

B: $g(t) \neq [_] \wedge t \in MengeTransitionenF\ddot{u}rGuard_C$.

O: $g(t)$ wird gelöscht.

Guard-Hinzufügen

B: $g(t) = [_] \wedge t \in MengeTransitionenF\ddot{u}rGuard_C$.

O: der Transition t wird ein Guard hinzugefügt, der zufällig aus G_C ausgewählt wird.

Vergleichsoperatoren-Ändern

B: $g(t) \neq [_] \wedge t \in MengeTransitionenF\ddot{u}rGuard_C$.

O: $g(t)$ wird nach Elementen aus der $MengeVergleichsoperatoren$ durchsucht. Falls ein

[10] Dies passiert genau dann, wenn im Guard und/oder im Effect Parameter des Triggers referenziert werden

solches Element gefunden wird, wird es durch ein davon verschiedenes Element, das zufällig aus der *MengeVergleichsoperatoren* ausgewählt wird, ersetzt. Z. B. wird der Vergleichsoperator $>$ durch einen der Operatoren aus der Menge $\{<=, <, >=\}$ ausgetauscht. Der Operator $==$ wird immer durch $!=$ ersetzt; umgekehrt gilt dies auch.

Boolesche-Operatoren-Ändern

B: $g(t) \neq [_] \wedge t \in MengeTransitionenFürGuard_C$.

O: die booleschen Operatoren im $g(t)$ werden geändert, d. h. && wird durch || ersetzt und umgekehrt.

Guard-Variable-Ändern

B: $g(t) \neq [_] \wedge t \in MengeTransitionenFürGuard_C$.

O: die im Guard referenzierten Variablen werden geändert. Für jede Variable wird untersucht, ob die Komponente, deren Verhalten durch den Zustandsautomat beschrieben wird, Variablen enthält, die den gleichen Typ wie die zu ändernde Variable haben. Falls solche Variablen existieren, dann wird eine entsprechende Variable zufällig ausgewählt und die zu ändernde Variable gegen die ausgewählte Variable ausgetauscht. Z. B. Guard im initialen Modell: $[a < 0]$, Guard im Mutant: $[b < 0]$. Somit können durch Anwendung dieses Operators mehrere Mutanten entstehen. Die Anzahl der erzeugten Mutanten hängt davon ab, wie viele Variablen im Guard referenziert werden und ob Variablen existieren, die als Ersatz in Frage kommen.

Effect-Operatoren

Variablenzuweisung-Operatoren

Variablenzuweisung-Löschen

B: $e(t) \in EV_C$.

O: $e(t)$ wird gelöscht.

Variablenzuweisung-Hinzufügen

B: $e(t) = [_]$.

O: der Transition t wird ein Effect hinzugefügt, der zufällig aus der Menge EV_C ausgewählt wird.

Arithmetischen-Operator-Ändern

B: $e(t) \in EV_C \lor e(t) \in EVA_C$.

O: der $e(t)$ wird nach Elementen aus der *MengeArithmetischeOperatoren* durchsucht. Falls ein solches Element gefunden wird, wird es durch ein davon verschiedenes Element, das zufällig aus der *MengeArithmetischeOperatoren* ausgewählt wird, ersetzt. So z. B. wird $+$ durch einen Operator aus der Menge $\{*, /, -\}$ ausgetauscht.

Bearbeitete-Variable-Ändern

B: $e(t) \in EV_C \lor e(t) \in EVA_C$.

O: die im $e(t)$ referenzierten Variablen werden geändert. Für jede Variable wird untersucht, ob die Komponente, deren Verhalten durch den Zustandsautomat beschrieben wird, Variablen enthält, die den gleichen Typ wie die zu ändernde Variable haben. Falls solche Variablen existieren, dann wird eine entsprechende Variable zufällig ausgewählt und die zu ändernde Variable gegen die ausgewählte Variable ausgetauscht. Z. B. Zuweisung im initialen Modell: $a := 0$, Zuweisung im Mutant: $b := 0$. Somit können durch Anwendung dieses Operators mehrere Mutanten entstehen. Die Anzahl der erzeugten Mutanten hängt davon ab, wie viele Variablen im Effect referenziert werden und ob Variablen vorhanden sind, die als Ersatz in Frage kommen.

Aufruf-Operatoren

Aufruf-Löschen

B: $e(t) \in EA_C$.

O: $e(t)$ wird gelöscht.

Aufruf-Hinzufügen

B: $e(t) = [_]$.

O: der Transition t wird ein Effect hinzugefügt, der zufällig aus EA_C ausgewählt wird.

Aufruf-Ändern

B: $e(t) \in EA_C \lor e(t) \in EVA_C$.

O: der im Effect vorhandene Aufruf einer Komponente wird durch einen anderen Aufruf derselben Komponente ersetzt.

Generische-Effect-Operatoren

Generischen-Effect-Löschen

B: $e(t) \in EVA_C$.

O: $e(t)$ wird gelöscht.

Generischen-Effect-Hinzufügen

B: $e(t) = [_]$.

O: der Transition t wird ein Effect hinzugefügt, der zufällig aus EVA_C ausgewählt wird.

Pre-state-Operatoren

Jede Transition eines Zustandsautomaten hat genau einen Pre-state. Im Unterschied zu Trigger, Guard oder Effect kann der Pre-state einer Transition nicht einfach gelöscht werden. Deshalb beinhaltet die Klasse der Pre-state-Operatoren nur den im Folgenden beschriebenen Operator.

Pre-state-Ändern: ändert den Pre-state einer Transition t, indem er diesen durch einen anderen Zustand des Zustandsautomaten ersetzt. Da dadurch auch nichtdeterministisches Verhalten erzeugt werden kann, sind die in Frage kommenden Zustände nur diejenigen, aus denen keine Transition existiert, die den gleichen Trigger wie die bearbeitete Transition t hat. Des Weiteren muss vor Anwendung des Operators gelten, dass der Pre-state der Transition mehr als eine ausgehende Kante hat. Anderenfalls würde die Mutation dazu führen, dass der alte Pre-state der Transition nach der Mutation keine ausgehenden Kanten mehr hat.

B: $|Ak_{pre(t)}| > 1$.

O: der Pre-state von t wird geändert; der neue Pre-state wird dabei zufällig aus der $MengeKandidatenPreState_t \setminus \{Anfangszustand_C\}$ ausgewählt.

Post-state-Operatoren

Ähnlich wie im Falle der Pre-state-Operatoren beinhaltet diese Klasse nur einen Operator, der den Post-state einer Transition ändert.

Post-state-Ändern: im Unterschied zur Änderung des Pre-states, kann durch Änderung des

Post-states einer Transition kein nichtdeterministisches Verhalten erzeugt werden. Deshalb beinhaltet die Menge der in Frage kommenden Zustände alle Zustände des Zustandsautomaten, außer dem Pre- und Post-state der aktuellen Transition t und dem Anfangszustand des Automaten.

B: $|Ek_{post(t)}| > 1$.

O: der Post-state von t wird geändert; der neue Post-state wird dabei zufällig aus der Menge $S_C \setminus \{\{post(t)\} \cup \{pre(t)\} \cup \{Anfangszustand_C\}\}$ ausgewählt.

Transition-Operatoren

Transition-Löschen: durch das Löschen einer Transition kann es vorkommen, dass bestimmte Zustände keine ein- oder ausgehenden Transitionen mehr besitzen. Deshalb wird der Operator auf der Transition t nur angewendet, wenn sowohl der Pre- als auch der Post-state der Transition mehr als eine ausgehende bzw. eingehende Transition haben.

B: $|Ak_{pre(t)}| > 1 \ \wedge \ |Ek_{post(t)}| > 1 \ \wedge \ tr(t) \neq [_]$.

O: Transition t wird gelöscht. Dies bedeutet auch, dass $tr(t)$, $g(t)$ und $e(t)$ gelöscht werden.

Transition-Hinzufügen

B: $pre(t) \neq post(t) \ \wedge \ tr(t) \neq [_]$.

O: ausgehend von einer Transition t wird eine neue Transition t_{neu} erzeugt, mit $tr(t_{\text{neu}}) = tr(t)$ und $post(t_{\text{neu}}) = post(t)$. Als $pre(t_{\text{neu}})$ wird ein Zustand zufällig aus der Menge $S_C \setminus \{\{pre(t)\} \cup \{Anfangszustand_C\}\}$ ausgewählt.

Zustand-Operatoren

Zustand-Löschen: dieser Operator betrachtet alle Zustände $S \in S_C$.

B: $|Ak_S| = 1 \ \wedge \ |Ek_S| = 1$.

O: der Zustand S wird gelöscht. Darüber hinaus muss die eingehende Transition t (mit $post(t) = S$) und die ausgehende Transition t_1 (mit $pre(t_1) = S$) angepasst werden. Der Post-state der Transition t wird gleich dem Post-state der Transition t_1 gesetzt, so dass: $post(t) = post(t_1)$. Daraufhin wird die Transition t_1 gelöscht.

Zustand-Hinzufügen: dieser Operator betrachtet alle Transitionen $t \in T_C$.

B: $|Ek_{post(t)}| > 1$.

O: ein neuer Zustand S wird erzeugt. Da für diesen Zustand mindestens jeweils eine eingehende und eine ausgehende Transition benötigt werden, wird als einzige eingehende Transition die Transition t gesetzt (so dass $post(t) = S$) und als einzige ausgehende Transition eine andere Transition t_1 aus der Menge $\{t_1 \in T_C | t_1 \neq t \wedge |Ak_{pre(t_1)}| > 1\}$ ausgewählt, so dass $pre(t_1) = S$.

Wie an der Beschreibung der einzelnen Modell-Mutationsoperatoren zu erkennen ist, ist die Granularität der eingefügten Änderungen unterschiedlich. So z. B. weicht ein Mutant, der durch den Operator Boolesche-Operatoren-Ändern generiert wurde, nur hinsichtlich der booleschen Operatoren (z. B. && statt ||) im Guard einer Transition eines Zustandsautomaten vom initialen Modell ab (z. B. Guard im initialen Modell: $[a > 0 \;\&\&\; b < 0]$; Guard im Mutant: $[a > 0 \;||\; b < 0]$). Andere Operatoren erzeugen dagegen größere Abweichungen, wie z. B. der Operator Transition-Löschen. Dieser löscht eine Transition und dabei gleichzeitig auch – falls vorhanden – den Trigger, den Guard und den Effect dieser Transition.

7.1.3. Fehlererkennungspotential der Testfallmengen für den Komponententest

Zur Bewertung des Fehlererkennungspotentials von Testfallmengen für den Komponententest, wurde die Mutationsanalyse für Testfallmengen durchgeführt, die jeweils die Kriterien Zustands-, Transitions- und Transitionspaarüberdeckung auf den Zustandsautomaten, die in Tabelle 6.2 beschrieben werden, erfüllen. Dafür wurden in einem ersten Schritt Mutanten dieser Zustandsautomaten generiert. Zur Generierung dieser Mutanten wurden alle Modell-Mutationsoperatoren, mit Ausnahme der Operatoren der Klasse Aufruf-Operatoren, angewendet. Diese Operatoren wurden nicht in Betracht gezogen, weil die dadurch erzeugten Mutanten intermodulare Fehler[11] simulieren, die nur die Interaktion von Komponenten betreffen. Somit wurden für jeden Zustandsautomat mehrere Mutanten generiert: 132 Mutanten durch Anwendung der Operatoren auf den Zustandsautomat Konto, 429 Mutanten durch Anwendung auf den Zustandsautomat InfotainmentGui und 780 Mutanten durch Anwendung auf den komplexesten Zustandsautomat Achstest.

[11] Siehe Abschnitt 7.1.2.1

Anschließend wurden die in Tabelle 6.3 beschriebenen Testfallmengen (die Komponententestkriterien auf den jeweiligen Zustandsautomaten entsprechen) auf diese Mutanten automatisch ausgeführt und der erreichte Mutation-score berechnet (gemäß der Formel 7.1, ohne Berücksichtigung der funktionsäquivalenten Mutanten, d. h. unter der Annahme $|E(MO)| = 0$). Die von den Testfallmengen erreichten Mutation-scores werden in Tabelle 7.1 in der Spalte *MS* beschrieben. Darüber hinaus werden in der Spalte *TC* die Anzahl der Testfälle und in der Spalte *UE* die Anzahl der durch die Testfallmenge überdeckten Entitäten angegeben. Ein Teil dieser Ergebnisse wurde bereits in [PS10] vorgestellt.

		Zustandsüb.			Transitionsüb.			Transitionspaarüb.		
ZM	AM	TC	UE	MS	TC	UE	MS	TC	UE	MS
Konto	132	2	8	23,48	6	19	68,18	22	58	74,24
InfotainmentGui	429	4	17	33,8	16	54	70,4	48	207	78,09
Achstest	780	8	26	23,97	27	83	55,51	80	272	63,97
Mittelwerte	447	4,67	17	27,08	16,33	52	64,7	50	179	72,1

ZM: Betrachteter Zustandsautomat
AM: Anzahl generierter Mutanten
TC: Anzahl generierter Testfälle
UE: Anzahl überdeckter Entitäten (Zustände, Transitionen oder Transitionspaare)
MS: Erreichter Mutation-score (in %)

Tabelle 7.1.: Ergebnisse der Mutationsanalyse für Komponententestfälle

Um auf die einzelnen Testfallmengen leichter Bezug zu nehmen, werden folgende Abkürzungen eingeführt: bezüglich des Zustandsautomaten Konto wird die Testfallmenge, die alle Zustände überdeckt als *ZK* (*Z* von Zustandsüberdeckung und *K* von Konto) bezeichnet, *TK* ist die Testfallmenge, die Transitionsüberdeckung erfüllt und *TPK* ist die Testfallmenge die Transitionspaarüberdeckung erfüllt. Ähnliche Abkürzungen werden für die Testsuites eingeführt, die InfotainmentGui und Achstest überdecken: hier wird das *K* durch *I* (*I* von InfotainmentGui) respektive *A* (*A* von Achstest) ersetzt. Als Beispiel: mit *TPI* ist die Testfallmenge gemeint, die Transitionspaarüberdeckung auf dem Zustandsautomat InfotainmentGui erreicht.

Von den neun betrachteten Testsuites, erreichte TPI mit 78,09% den höchsten Mutation-score, gefolgt von der Testsuite TPK. Bei Betrachtung der Ergebnisse pro Modell fällt auf, dass die Testfallmengen für Transitionspaarüberdeckung immer besser abschneiden, als die anderen zwei Testfallmengen, die auf Grundlage desselben Modells generiert wurden und die Kriterien Transitionsüberdeckung bzw. Zustandsüberdeckung erfüllen. Auch im Falle des Zustandsautomaten Achstest trifft das zu, obwohl TPA nur 97,84% der machbaren Transitionspaare überdeckt. Im Folgenden werden diese Ergebnisse pro Modell etwas genauer beleuchtet.

Die Testsuite ZK enthält lediglich 2 Testfälle, die 23,48% also insgesamt 31 Mutanten zu erkennen erlauben. Die Testsuite TK enthält 6 Testfälle und entdeckt 90 Mutanten, was 59 mehr bedeutet als ZK. Die Testfallmenge TPK beinhaltet mit 22 Testfällen 16 Testfälle mehr als TK, entdeckt allerdings lediglich 8 zusätzliche Mutanten.

Die Testsuite ZI mit 4 Testfällen ermöglicht die Erkennung von 145 Mutanten, was 33,8% aller Mutanten entspricht. Die Testfallmenge TI enthält 12 Testfälle mehr als ZI, wodurch 157 zusätzliche Mutanten erkannt werden. Beim Vergleich der Testfallmengen TI und TPI fällt ein noch größerer Unterschied in der Testfallanzahl auf: TPI enthält 32 Testfälle mehr als TI. Dadurch entdeckt TPI 33 Mutanten mehr als TI.

Mit 8 Testfällen erkennt die Testsuite ZA 187 Mutanten, die aus dem Zustandsautomat Achstest generiert wurden. 19 Testfälle mehr enthält die Testsuite TA, erlaubt dadurch die Erkennung von 246 zusätzlichen Mutanten. Die 80 Testfälle der Testsuite, die Transitionspaarüberdeckung zu 97,84% auf dem Zustandsautomat Achstest erreicht, erlauben es 66 mehr Mutanten zu entdecken als die Testsuite für Transitionsüberdeckung TA.

Im Durchschnitt erlaubte die Transitionspaarüberdeckung die Erkennung von 72,1% der generierten Mutanten, gefolgt von der Transitionsüberdeckung, die 64,7% erkannte. Am schlechtesten hat die Zustandsüberdeckung abgeschnitten, die im Durchschnitt nur die Erkennung von 27,08% der generierten Mutanten ermöglichte. Aufgrund der Tatsache, dass von den drei betrachteten Kriterien die Zustandsüberdeckung das schwächste und die Transitionspaarüberdeckung das stärkste ist, sind diese Ergebnisse nicht überraschend.

Die Erklärung für das schlechte Abschneiden der Zustandsüberdeckung hängt mit dem Aufbau der Zustandsautomaten und der Art der durch dieses Kriterium geforderten Überdeckung zusammen. Zum einen gilt für die Zustände eines Zustandsautomaten, dass außer dem Anfangszustand und dem Endzustand alle Zustände beliebig viele ein- und ausgehende Transitionen haben kön-

nen[12]. Zum anderen werden von all den Transitionen eines Zustandsautomaten durch Testfallmengen, die Zustandsüberdeckung erreichen, im Regelfall nur ein Teil der ein- und ausgehenden Transitionen der überdeckten Zustände durchlaufen. In jedem Fall werden solche Testsuites mindestens eine eingehende Transition jedes Zustands überdecken. Solche Testsuites überdecken also mindestens so viele Transitionen, wie die Anzahl der Zustände des Zustandsautomaten. Dies ist deshalb relevant, da eine notwendige Bedingung für die Erkennung eines Mutanten besagt, dass die mutierte Entität (Transition oder Zustand) von der Testfallmenge überdeckt werden muss.

Zusammenfassend kann gesagt werden, dass der Grund für das schlechte Abschneiden der Zustandsüberdeckung der ist, dass durch Testfallmengen, die alle Zustände des Automaten überdecken, viele Transitionen nicht überdeckt werden. Somit können Mutanten, die eine Abweichung an einer nicht überdeckten Transition enthalten, gar nicht erkannt werden.

Deutlich besser als die Zustandsüberdeckung hat die Transitionsüberdeckung abgeschnitten. Testfallmengen für Transitionsüberdeckung erreichen im Schnitt einen Mutation-score von 64,7%. Damit erkennen sie im Schnitt 37,62% mehr Mutanten als Testfallmengen für Zustandsüberdeckung und liegen im Schnitt nur 7,4% hinter dem von der Transitionspaarüberdeckung erreichten Mutation-score zurück. Dies liegt daran, dass Testfallmengen für Transitionsüberdeckung alle mutierten Entitäten erreichen[13].

Auch die Testfallmengen für Transitionspaarüberdeckung erreichen alle mutierten Entitäten. Im folgenden Unterabschnitt werden die Arten der erkannten Mutanten genauer betrachtet. Dadurch wird aufgezeigt, weshalb Testfallmengen für Transitionspaarüberdeckung besser abschneiden, als Testfallmengen für Transitionsüberdeckung.

[12] Die Einschränkungen beim Anfangs- und Endzustand lauten wie folgt: der Anfangszustand darf eine ausgehende Transition haben, aber keine eingehenden. Der Endzustand darf beliebig viele eingehenden, aber keine ausgehenden Transitionen haben

[13] Modell-Mutationsoperatoren bearbeiten entweder einen Zustand oder eine Transition; eine Testfallmenge, die alle Transitionen überdeckt, überdeckt auch alle Zustände

7.1.3.1. Vergleich der Fehlerarten

Für den Zweck des Vergleichs der erkennbaren Fehlerarten wurde die Fehlererkennbarkeit bezüglich der sieben Mutationsklassen[14] auf den drei Zustandsautomaten einzeln betrachtet. Abbildung 7.3 stellt die Ergebnisse der Mutationsanalyse pro Mutationsklasse auf dem Modell Konto dar. Die Testfallmenge ZK schneidet mit einer Ausnahme bezüglich aller Mutationsklassen schlecht bis sehr schlecht ab. Mutanten der Klasse Effect-Mutationen werden gar nicht durch diese Testfallmenge erkannt. TK und TPK schneiden auf diesem Modell bezüglich vier der sieben Klassen ähnlich gut ab: alle Mutanten der Klassen Pre-state-, Post-state- und Zustand-Mutationen werden durch diese zwei Testfallmengen erkannt, bezüglich Trigger-Mutationen wird ein Mutation-score von 87,5% erreicht. Ähnlich schlecht schneiden beide Testfallmengen bezüglich Guard- (42,31% der Guard-Mutanten werden erkannt) und Transition-Mutationen (55,17% der Transition-Mutanten werden erkannt) ab. Lediglich hinsichtlich der Klasse Effect-Mutationen kann sich TPK absetzen, da sie circa doppelt so viele Effect-Mutanten erkennt wie TK.

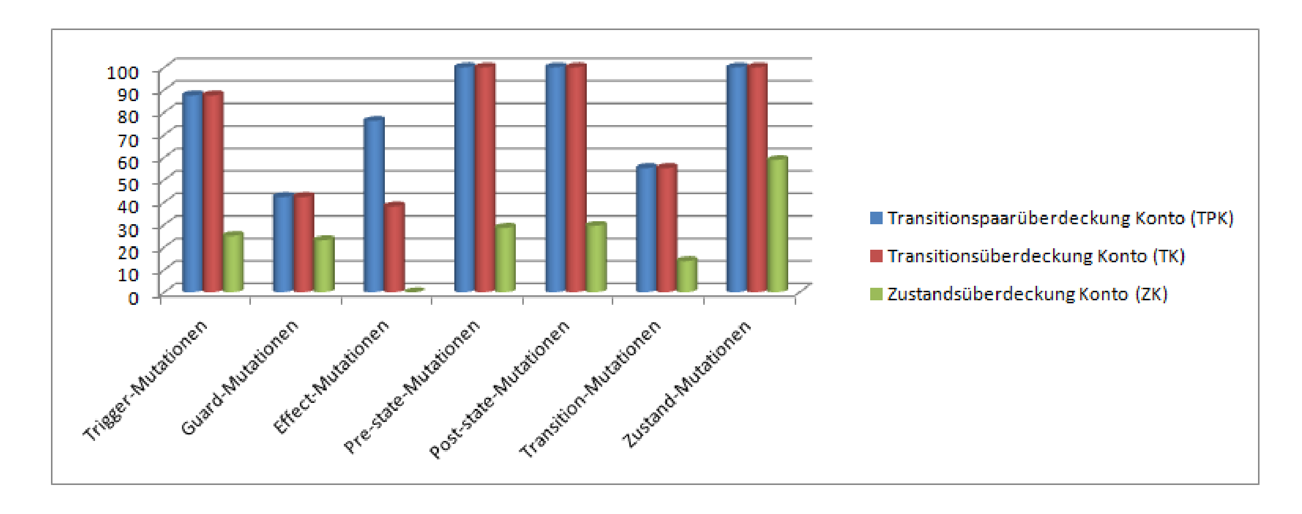


Abbildung 7.3.: Fehlererkennungspotential bzgl. einzelner Mutationsklassen von Testfallmengen, die Komponententestkriterien auf der Komponente Konto erfüllen

Ähnliche Ergebnisse werden auch anhand des Zustandsautomaten InfotainmentGui erreicht; diese werden in Abbildung 7.4 dargestellt. Einzig bezüglich der Klasse Zustand-Mutationen er-

[14] Eine Mutationsklasse beinhaltet alle Mutationen, die von Modell-Mutationsoperatoren einer bestimmten Klasse erzeugt wurden. So z. B. beinhaltet die Mutationsklasse Trigger-Mutationen alle Mutanten, die durch Modell-Mutationsoperatoren der Klasse Trigger-Operatoren erzeugt wurden

reicht die Testfallmenge ZI gute Ergebnisse, da sie 75,51% der Mutanten dieser Klasse erkennt. Bezüglich Effect-Mutationen schneidet ZI auch auf diesem Modell sehr schlecht ab, da nur 1,1% der generierten Effect-Mutanten erkannt werden. Die Testfallmengen TI und TPI schneiden bezüglich drei der sieben Klassen gleich gut ab. Post-state- und Zustand-Mutationen werden zu 100% erkannt, Pre-state-Mutationen zu 98%. Bezüglich der Klasse Trigger- und Guard-Mutationen schneidet TPI etwas besser ab als TI: TPI erkennt 3,63% mehr Trigger- und 10,2% mehr Guard-Mutanten. Ähnlich wie auf dem Modell Konto werden durch TPI fast doppelt so viele Effect-Mutanten im Vergleich zu TI erkannt. Bezüglich Transition-Mutationen schneiden sowohl TPI als auch TI schlecht ab (49,43%).

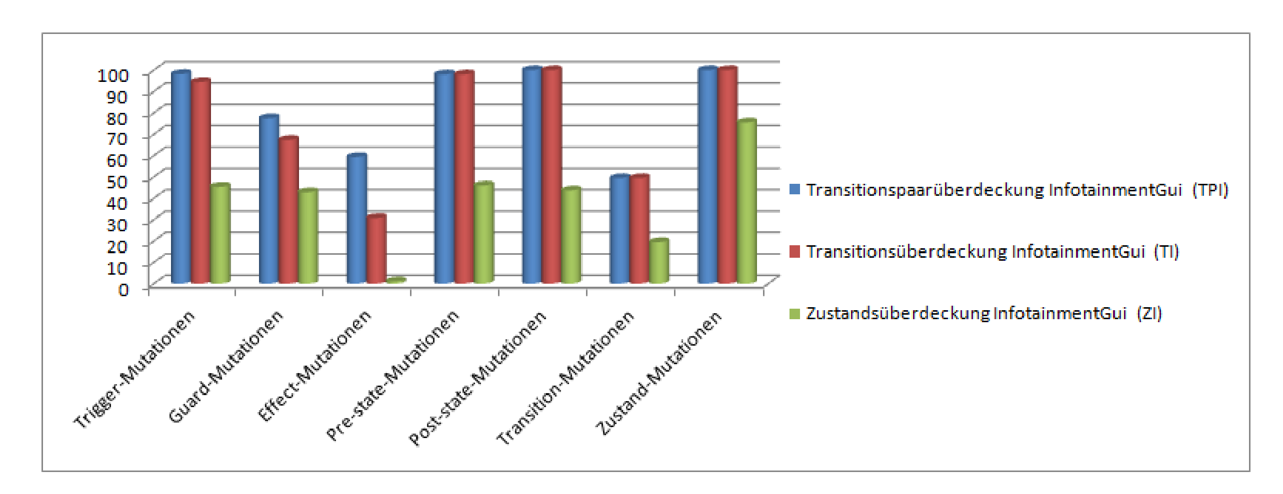


Abbildung 7.4.: Fehlererkennungspotential bzgl. einzelner Mutationsklassen von Testfallmengen, die Komponententestkriterien auf der Komponente InfotainmentGui erfüllen

Nachdem auf den einfacheren Modellen die Transitionsüberdeckung und die Transitionspaarüberdeckung mit Ausnahme der Effect-Mutationen relativ ähnlich abschneiden, ergibt sich für das komplexeste Modell ein etwas anderes Bild. Dies wird in Abbildung 7.5 dargestellt. Diese Ergebnisse wurden bereits in [PS10] veröffentlicht. Die Testfallmenge TPA schneidet im Vergleich zu TA auf allen Mutationsklassen besser ab. Der größte Unterschied entsteht auch hier bei den Effect-Mutanten. Trotz dieses Unterschiedes ist der durch TPA erreichte Mutation-score bezüglich dieser Mutationsklasse schlecht, da lediglich 32,43% der Effect-Mutationen erkannt werden. Die Zustandsüberdeckung schneidet auch auf diesem Modell sehr schlecht ab; lediglich bezüglich Zustand-Mutationen werden mehr als 50% der erzeugten Mutanten erkannt (51,95%).

Zusammenfassend können folgende Bemerkungen zu der Art erkennbarer Fehler gemacht

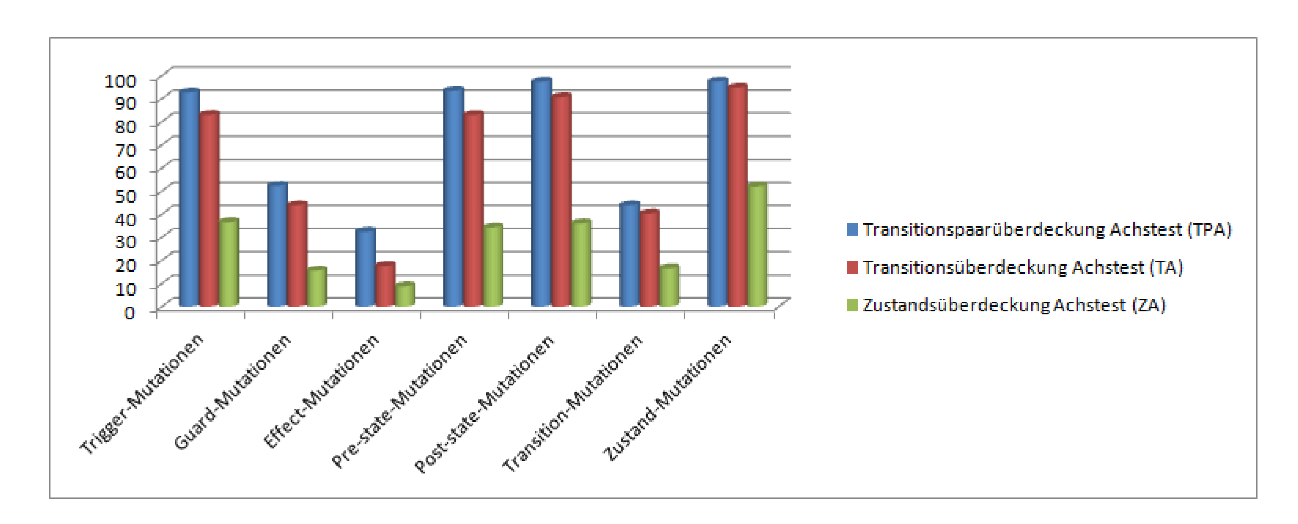


Abbildung 7.5.: Fehlererkennungspotential bzgl. einzelner Mutationsklassen von Testfallmengen, die Komponententestkriterien auf der Komponente Achstest erfüllen (aus [PS10])

werden: nachdem Testfallmengen für Zustandsüberdeckung generell einen sehr niedrigen Mutation-score erreichen, überrascht es nicht, dass sie auch bezüglich der einzelnen Mutationsklassen schlecht abschneiden. Nur bezüglich Zustand-Mutationen erreichen sie Mutation-scores, die über 50% liegen (auf InfotainmentGui sogar 75,51%). Somit eignen sich Testfallmengen, die Zustandsüberdeckung erfüllen, bestenfalls zur Erkennung einiger Zustandsfehler.

Im Gegensatz dazu, liefert die Transitionsüberdeckung bezüglich vier der sieben Mutationsklassen gute bis sehr gute Ergebnisse, was dazu führt, dass sie insgesamt betrachtet nicht viel schlechter als die Transitionspaarüberdeckung abschneidet. Somit eignen sich Testfallmengen, die alle Transitionen oder alle Transitionspaare eines Automaten überdecken gut für die Erkennung von Trigger-Fehlern, Pre-state-Fehlern, Post-state-Fehlern und Zustand-Fehlern.

Bezüglich der Mutationsklassen Guard- und Transition-Mutationen schneiden die Transitions- und die Transitionspaarüberdeckung ähnlich schlecht ab. Die einzige Klasse bei der sich die Transitionspaarüberdeckung deutlich von der Transitionsüberdeckung absetzen kann, ist die Klasse Effect-Mutationen. Hier entdeckt die Transitionspaarüberdeckung im Schnitt circa doppelt so viele Mutanten, wie die Transitionsüberdeckung. Trotzdem ist der erreichte Mutation-score bezüglich dieser Mutationsklasse niedrig. Im Folgenden werden die drei Mutationsklassen, bezüglich deren die Transitionspaarüberdeckung schlecht abschneidet, etwas genauer betrachtet.

Eine dieser Mutationsklassen ist die Klasse Effect-Mutationen. Der Grund, weshalb Effect-

Mutanten nur zu einem niedrigen Anteil erkannt werden ist, dass die Änderung eines Effects $e(t)$ keine Auswirkung auf das Durchlaufen der betroffenen Transition t hat. Falls die Transition t im initialen Modell durchlaufen wird, dann wird sie auch im Mutant durchlaufen. Diese Änderungen beeinträchtigen nur das Durchlaufen anderer Transitionen, die im Guard die Variable (die im geänderten Effect bearbeitetet wird) referenzieren. Um überhaupt die Möglichkeit zu haben, einen solchen Mutanten zu erkennen, müsste ein Testfall also die Transition t in Kombination mit anderen Transitionen überdecken, die entsprechende Guards beinhalten. Das Kriterium Transitionspaarüberdeckung ist aber nicht auf so eine Überdeckung ausgerichtet. Für die Erkennung solcher Mutanten könnten sich Testfallmengen eignen, die *datenflussbasierte Überdeckungskriterien* auf Modellebene erfüllen.

Ähnlich schlecht wie gegenüber Effect-Mutationen schneidet die Transitionspaarüberdeckung bezüglich Guard-Mutationen ab. Die zur Transitionspaarüberdeckung generierten Testfälle sind so generiert, dass bei der Ausführung der Testfälle auf dem initialen Modell die Guards der traversierten Transitionen der Zustandsautomaten zu wahr ausgewertet werden. Somit haben hauptsächlich die Mutanten eine Chance erkannt zu werden, die im Vergleich zum initialen Modell eine strengere Guard-Bedingung enthalten (Dies wird z. B. erreicht indem der Oder-Verknüpfungsoperator || durch den Und-Verknüpfungsoperator && ersetzt wird). Da dies aber im Allgemeinen nicht der Fall ist, werden auch Guard-Mutationen zu einem geringen Anteil erkannt. Zur Erkennung solcher Mutanten könnten sich Testfallmengen eignen, die *Bedingungsüberdeckungskriterien* auf Modellen erfüllen.

Auch bezüglich der Mutationsklasse Transition-Mutationen erreicht die Transitionspaarüberdeckung niedrige Mutation-scores (55,17% auf Konto; 49,43% auf InfotainmentGui und 43,88% auf Achstest). Zu der Klasse Transition-Mutationen gehören solche Mutanten, die sich von dem initialen Modell dadurch unterscheiden, dass sie entweder eine Transition mehr oder eine weniger enthalten. Nicht erkennbar sind Mutanten, die eine Transition mehr als das initiale Modell beinhalten: eine aus dem intialen Modell generierte Testfallmenge wird die im Mutant zusätzlich vorhandene Transition nicht überdecken, weil nur die Transitionen des Mutanten überdeckt werden, die auch im initialen Modell vorhanden sind. Das führt dazu, dass ein solcher Mutant nicht erkannt wird. Dagegen gut erkennbar sind Mutanten, die im Unterschied zum originalen Modell eine Transition weniger enthalten. Da durch beide Operatoren auf den drei Modellen ähnlich viele Mutanten erzeugt wurden und die Transition-Hinzufügen Mutanten[15] im Gegen-

[15]Sind Mutanten die durch den Modell-Mutationsoperator Transition-Hinzufügen generiert wurden

satz zu den Transition-Löschen Mutanten[16] nicht erkannt werden, werden etwa die Hälfte der Transition-Mutationen erkannt.

Allgemein lässt sich feststellen, dass das Fehlererkennungspotential von Komponententestfällen – über die Testüberdeckungsmaße hinaus – wesentlich von der Art der injizierten Mutationen abhängt. Es ist nicht überraschend, dass diejenigen Fehlerarten am ehesten erkannt wurden, die an den zu überdeckenden Modellentitäten angelehnt waren, etwa Trigger-Fehler oder Zustand-Fehler. Andere Fehlerarten dagegen, wie z. B. Effect-Fehler oder Guard-Fehler, die nicht explizit Gegenstand der Modellüberdeckung sind, konnten mit geringeren Häufigkeiten aufgedeckt werden.

7.1.4. Fehlererkennungspotential der Testfallmengen für den Integrationstest

Das implementierte Mutationsmodul ermöglicht auch die Untersuchung der Fehlererkennung von Testfällen, die zustandsbasierten Integrationstestkriterien genügen. Dazu wurden zunächst Mutanten generiert, die fehlerhafte Interaktionen zwischen Komponenten simulieren. Zur Erzeugung solcher Mutanten wurden nicht alle Modell-Mutationsoperatoren angewendet. Zum einen wurden die Transition- und Zustand-Operatoren ausgelassen. Aus der Klasse der Effect-Operatoren wurden die Variablenzuweisung-Operatoren nicht eingesetzt. Die generierten Effect-Mutanten, die im Folgenden beschrieben werden, unterscheiden sich also alle hinsichtlich der Aufrufe in den Effects vom initialen Modell. Des Weiteren wurden die Operatoren Trigger-Hinzufügen, Aufruf-Hinzufügen, Generischen-Effect-Hinzufügen und Guard-Hinzufügen nicht angewendet. Diese wurden weggelassen, um keine neuen Mappings im Modell zu erzeugen bzw. um zu vermeiden, dass bestimmte Mappings, die im initialen Modell ausführbar sind, im Mutant nicht ausführbar sind.

Die Modell-Mutationsoperatoren wurden nicht auf alle Transitionen der interagierenden Zustandsmaschinen angewendet, sondern nur auf die Transitionen, die zu Mappings gehören. Dadurch entstanden deutlich weniger Mutanten als bei der Mutantengenerierung aus einzelnen Zustandsmaschinen. Es wurden 72 Mutanten aus dem Modell CabinControl, 94 aus dem Modell Infotainment und 263 Mutanten aus dem Modell Liegenprüfplatz erzeugt. Daraufhin wurden die in Tabelle 6.6 beschriebenen Testsuites, die zustandsbasierte Überdeckungskriterien erfüllen, auf

[16]Sind Mutanten die durch den Modell-Mutationsoperator Transition-Löschen generiert wurden

die Mutationen des Modells ausgeführt und der Mutation-score berechnet. Die Ergebnisse werden in Tabelle 7.2 beschrieben. Ein Teil dieser Ergebnisse wurde bereits in [PS10] vorgestellt.

Wie schon in Abschnitt 6.4 erwähnt, gilt, dass zwei oder mehrere zustandsbasierte Überdeckungskriterien auf bestimmten Modellen die gleiche Überdeckung fordern können. Dies trifft unter anderem für die Kriterien *Pre-states and triggers* und *Pre-states, triggers and effects* bezüglich der Modelle Infotainment und Liegenprüfplatz zu. In diesem Fall wurden nicht zwei Testfallmengen generiert, die beide die gleiche Überdeckung erreichen, sondern nur eine Testfallmenge, die beide Kriterien erfüllt. Da es eine einzige Testfallmenge für beide Kriterien gibt, ist der Mutation-score der beiden Überdeckungskriterien gleich, was in den entsprechenden Zellen durch „$\rightarrow$“ angegeben wird.

M	AM	K1	K2	K3	K4	K5	K6	K7	K8
CabinControl	72	73,61	$\rightarrow$	77,78	86,11	84,72	$\rightarrow$	95,83	98,61
Infotainment	94	$\rightarrow$	82,98	93,62	47,87	72,34	87,23	$\rightarrow$	93,62
Liegenprüfplatz	263	$\rightarrow$	66,92	74,52	56,27	72,24	77,95	83,27	84,03
Mittelwerte	143	74,5	75,89	81,97	63,42	76,43	87	90,9	92,08

M: Betrachtetes Modell
AM: Anzahl generierter Mutanten
K1–K8: siehe Tabelle 6.5

Tabelle 7.2.: Ergebnisse der Mutationsanalyse für Integrationstestfälle

Um auf die einzelnen Testfallmengen im Text leichter Bezug nehmen zu können, werden hier Abkürzungen für jede analysierte Testfallmenge eingeführt. *K1C* referenziert die Testfallmenge, die auf dem Modell CabinControl das Überdeckungskriterium *Pre-states and triggers* erfüllt; *K8C*, die Testfallmenge, die das Kriterium *Invoking / invoked transitions* auf dem Modell CabinControl erfüllt[17]. Analog werden für die anderen Testfallmengen auch die Abkürzungen *K1I* bis *K8I* (*I* von Infotainment) und *K1L* bis *K8L* (*L* von Liegenprüfplatz) verwendet.

Am besten schneidet die Testfallmenge K8C ab. Sie erlaubt die Erkennung von 98,61% aller anhand von CabinControl generierten Mutanten. Dafür wurden 5 Testfälle benötigt. Die Testfallmenge K7C erkennt 2 Mutanten weniger und enthält 3 Testfälle. Auf CabinControl hat die

[17]Für die Bedeutung von K1–K8: siehe Tabelle 6.5

Testfallmenge K1C den geringsten Mutation-score (73,61%), da sie 19 Mutanten nicht erkennt. Dies bedeutet, dass auf diesem Modell das Kriterium *Pre-states and triggers* das schwächste Fehlererkennungspotential hat.

Die Mutationsanalyse anhand des Infotainment-Modells ergab, dass die Testfallmenge K8I am besten abschneidet. Sie entdeckt 88 Mutanten, was 93,62% der generierten Mutanten entspricht und enthält 5 Testfälle. Auf diesem Modell ist *Effects on pre-states* mit Abstand das schwächste Kriterium, da die Testfallmenge K4I nur 47,87% der Mutanten erkennt. Dieser Wert liegt mehr als 20% unter dem Wert, der durch die Testfallmenge K5I erreicht wird. Bei Betrachtung der von den zwei zugrunde liegenden Kriterien geforderten Überdeckung wird klar, wieso das so ist: das Kriterium *Effects on pre-states* fordert auf diesem Modell die Überdeckung von mindestens 5 ausführbaren Mappings. Dagegen fordert das Kriterium *Effects in / on pre-states* die Überdeckung von mindestens 14 ausführbaren Mappings.

Auch auf dem Liegenprüfplatz-Modell erkennt die Testfallmenge, die das anspruchsvollste Kriterium erfüllt, die meisten Mutanten: K8L erkennt 221 Mutanten, was 2 Mutanten mehr sind als die Testfallmenge K7L. Auf diesem Modell wird die schwächste Überdeckung vom Kriterium *Effects on pre-states* gefordert, da das Kriterium lediglich 21 Mappings zu überdecken verlangt. Deshalb überrascht es nicht, dass die Testfallmenge K4L den niedrigsten Mutation-score erreicht.

Wie auch im Falle der Komponententestkriterien lässt sich generell feststellen, dass je anspruchsvoller das zugrunde liegende Kriterium ist, desto mehr Mutanten werden durch Testfallmengen, die dieses Kriterium erfüllen, erkannt. So erkennt pro Modell immer die Testsuite, die das anspruchsvollste Integrationstestkriterium erfüllt, auch die meisten Mutanten. Im Durchschnitt erkennen Testfallmengen, die das Kriterium *Invoking / invoked transitions* erfüllen, 92,08% der Mutanten, dicht gefolgt von Testfallmengen, die das Kriterium *Invoking transitions on pre-states* erfüllen, die im Schnitt 90,9% der Mutanten erkennen. Am schlechtesten abgeschnitten hat das Kriterium *Effects on pre-states*, da durch Testfallmengen, die dieses Kriterium erfüllen, im Schnitt nur 63,42% der Mutanten erkannt werden.

Schlussfolgernd kann gesagt werden, dass sich die Subsumptionshierarchie der Kriterien auch in den Mutation-scores widerspiegelt. Eine Testfallmenge, die ein bestimmtes Kriterium K erfüllt, entdeckt tendenziell mehr oder zumindest gleich viele Mutanten wie eine Testfallmenge, die ein von K subsumiertes Kriterium erfüllt. Die einzige Ausnahme stellen die Testfallmengen K4C und K5C dar. Die Testfallmenge K4C erkennt 62 und damit 86,11% der anhand von CabinControl generierten Mutanten. Damit erkennt sie einen Mutanten mehr als K5C.

Um den Grund für dieses unerwartete Ergebnis darzustellen, wird hier erst einmal auf die Abbildung 7.6 verwiesen, welches ein initiales Modell *IM* mit zwei Zustandsautomaten beschreibt. Im Unterschied zu dem in Abbildung 7.1 beschriebenen Modell, enthält Transition *tA7* denselben Aufruf an Komponente *B* wie Transition *tA6*. Somit gibt es in diesem Modell insgesamt vier Mappings. Sei der Mutant namens *MutIM* ein Modell, dass im Unterschied zum initialen Modell *IM* keinen Aufruf in *tA6*[18] enthält. Nun wird gezeigt, dass es zwei Testfälle geben kann, die die Transition *tA6* (und auch das Mapping *m1=(tA6, tB4)*) überdecken, sich aber hinsichtlich der Erkennbarkeit des Mutanten *MutIM* unterscheiden.

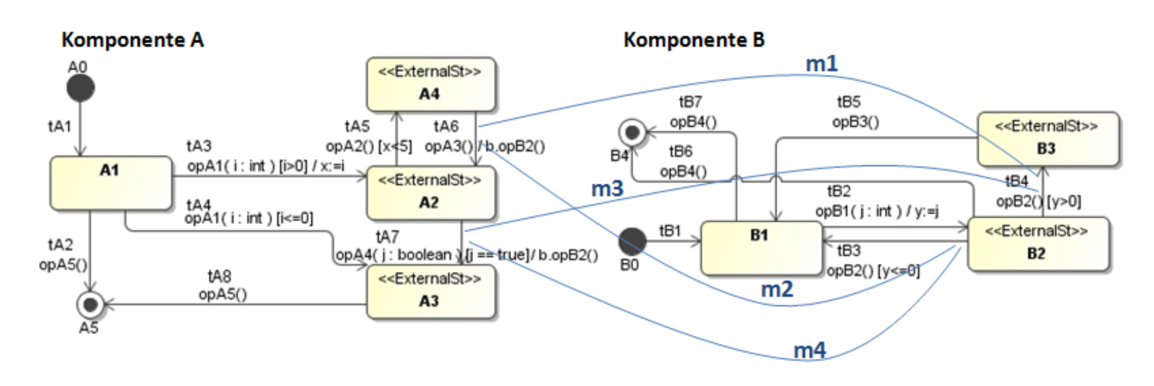


Abbildung 7.6.: Beispiel für ein erweitertes Modell *IM* mit zwei interagierenden Komponenten und mit vier Mappings

Der Testfall *T1: opA1(4), opB1(69171531), opA2(), opA3()* erkennt diesen Mutanten, da sich bei der Ausführung dieses Testfalls die externen Pfade durch Komponente *B* auf dem Mutant und auf dem initialen Modell unterscheiden. Die externen Pfade durch A bleiben gleich. Die bei Ausführung des Testfalls *T1* durchlaufenen externen Pfade durch *B* sind:

externer Pfad durch *B* im Modell *IM*: *B2, B3*

externer Pfad durch *B* im Modell *MutIM*: *B2*

Der Testfall *T2: opA1(4), opB1(69171531), opA2(), opA3(), opA4(true)* erkennt diesen Mutanten nicht, da sich bei der Ausführung dieses Testfalls die externen Pfade durch Komponente *A* und *B* auf dem Mutant und auf dem initialen Modell gleichen. Die bei Ausführung des Testfalls *T2* durchlaufenen externen Pfade sind:

externer Pfad durch *A* im Modell *IM* und im Mutant *MutIM*: *A2, A4, A2, A3*

externer Pfad durch *B* im Modell *IM* und im Mutant *MutIM*: *B2, B3*

[18]Solch ein Mutant kann durch den Operator Aufruf-Löschen erzeugt werden

Nun zurück zur Erklärung, weshalb K4C besser Abschneidet als K5C. Dazu wurde die Erkennbarkeit dieser zwei Testsuites bezüglich einzelner Mutationsklassen genauer betrachtet. Dabei stellte sich heraus, dass bezüglich der Mutanten, die durch den Operator Aufruf-Löschen erzeugt wurden, K4C besser abschneidet als K5C, da sie zusätzlich zu den von K5C erkannten Mutanten noch einen Mutanten M erkennt. Beide Testfallmengen überdecken die Transition t, an der sich der Mutant M vom initialen Modell unterscheidet. Dass K5C diesen Mutanten nicht erkennt kommt daher, da K5C einen Testfall enthält, der neben der Transition t zusätzliche Transitionen überdeckt, die denselben Aufruf wie t an die aufgerufene Komponente enthalten. Somit wird das Fehlen des Aufrufs an der Transition t verdeckt (also eine ähnliche Situation wie beim vorhin beschriebenen Testfall T2), was dazu führt, dass bei der Ausführung dieses Testfalls (aus K5C) auf dem initialen Modell und auf dem Mutanten derselbe externe Pfad in der aufgerufenen Komponente durchlaufen wird. Somit entdeckt K5C diesen Mutanten nicht.

Ein weiteres unerwartetes Ergebnis ist, dass die Testfallmenge K3I den gleichen Mutation-score erreicht, wie die Testfallmenge K8I. Da der Testfallmenge K8I ein anspruchsvolleres Überdeckungskriterium zu Grunde liegt, überdeckt es alle Mappings, die auch von K3I überdeckt werden und zusätzlich dazu drei weitere Mappings. Die von K8I überdeckten zusätzlichen Mappings sind nicht ausschlaggebend, da sie keine Transitionen enthalten, die nicht in den von K3I überdeckten Mappings enthalten sind. Dies führt in diesem konkreten Fall dazu, dass die erreichten Mutation-scores der beiden Testfallmengen gleich sind.

7.1.4.1. Vergleich der Fehlerarten

In diesem Abschnitt werden die Arten der Fehler genauer beleuchtet, die durch Integrationstestfälle entdeckt werden können. Ähnlich wie im Falle der Komponententestfälle wurde dazu eine Analyse bezüglich jeder Mutationsklasse durchgeführt. Im Unterschied zu der Analyse aus Abschnitt 7.1.3 werden hier fünf Mutationsklassen betrachtet, da die Modell-Mutationsoperatoren der Klasse Transition-Operatoren und Zustand-Operatoren nicht angewendet wurden.

Die von den Testfallmengen K1C bis K8C erreichten Ergebnisse pro Mutationsklasse werden in Abbildung 7.7 dargestellt. Beim Vergleich mit den in Abbildung 7.8 und 7.9 beschriebenen Ergebnissen fällt auf, dass die Mutationsklasse Guard-Mutationen fehlt. Der Grund dafür ist, dass durch die Anwendung von Guard-Operatoren auf diesem Modell keine Mutanten generiert wurden.

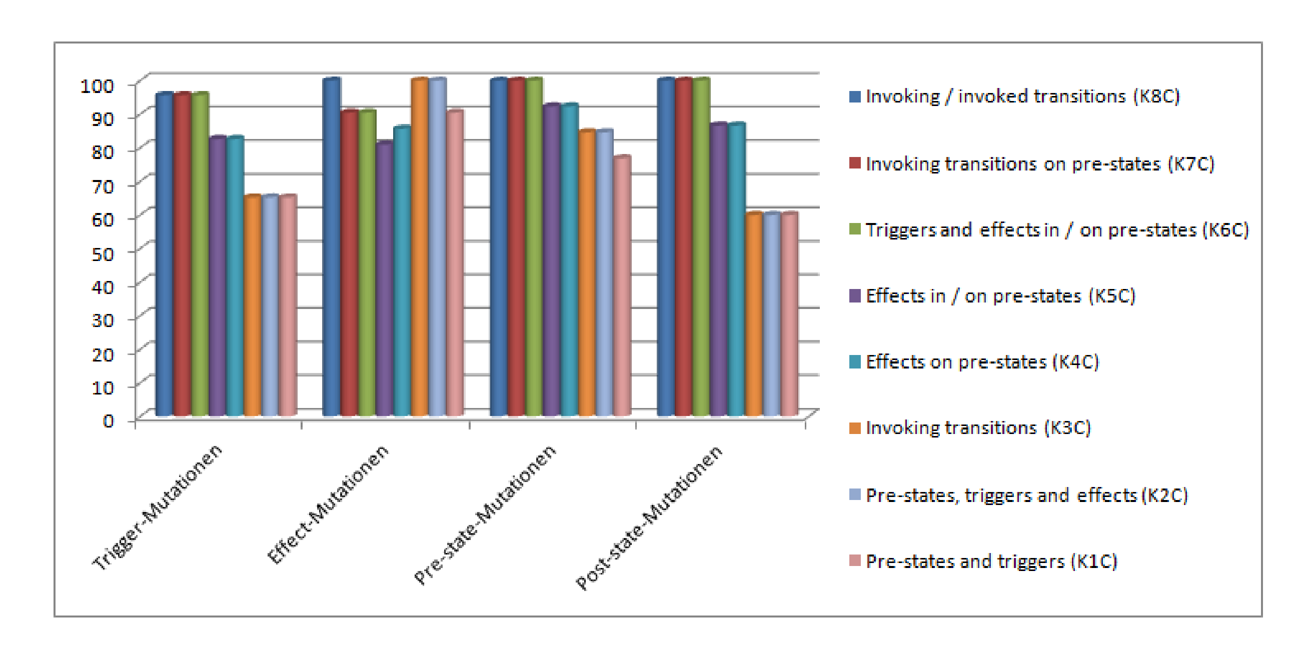


Abbildung 7.7.: Fehlererkennungspotential bzgl. einzelner Mutationsklassen von Testfallmengen, die zustandsbasierte Integrationstestkriterien auf dem Modell CabinControl erfüllen

Die Testfallmenge K8C erreicht bezüglich aller vier Mutationsklassen sehr gute Ergebnisse. Wie schon vorhin beschrieben, erreicht die Testfallmenge K4C einen höheren Mutation-score als die Testfallmenge K5C. In Abbildung 7.7 ist zu sehen, wo genau der Unterschied zwischen K4C und K5C entsteht, und zwar durch die unterschiedliche Erkennbarkeit bezüglich der Mutationsklasse Effect-Mutationen[19]. Des Weiteren ist anhand der Graphik zu erkennen, dass bezüglich Effect-Mutationen K3C besser abschneidet als K7C, obwohl das Kriterium *Invoking transitions on pre-states* das Kriterium *Invoking transitions* subsumiert. Dies lässt sich ähnlich erklären wie der Unterschied zwischen K4C und K5C. Insgesamt erreicht K3C aber einen niedrigeren Mutation-score als K7C, da durch K3C unter anderem die Pre-state- und Post-state-Mutationen zu einem kleineren Anteil erkannt wurden. Aufgrund der von *Invoking transitions on pre-states* geforderten Überdeckung, die auch die Pre-states auf der aufgerufenen Komponente betrachtet, ist dieses Ergebnis nicht überraschend.

Die detaillierten Ergebnisse der Mutationsanalyse für die Testfallmengen, die anhand des Modells Infotainment generiert wurden, werden in Abbildung 7.8 dargestellt. Wie schon vorhin erwähnt, erkennt K3I genau die gleichen Mutanten wie K8I, was zur Folge hat, dass diese zwei

[19]Die Mutanten, die durch den Operator Aufruf-Löschen erzeugt wurden, gehören zur Klasse der Effect-Mutationen

Testfallmengen auch bei der Betrachtung pro Mutationsklasse identische Ergebnisse aufweisen. Bezüglich vier der fünf Klassen schneiden beide sehr gut ab, da alle Mutationen dieser vier Klassen entdeckt werden. Die einzige Ausnahme stellt die Klasse Guard-Mutationen dar, da bezüglich dieser mit einer Ausnahme alle Testfallmengen gleich schlecht abschneiden.

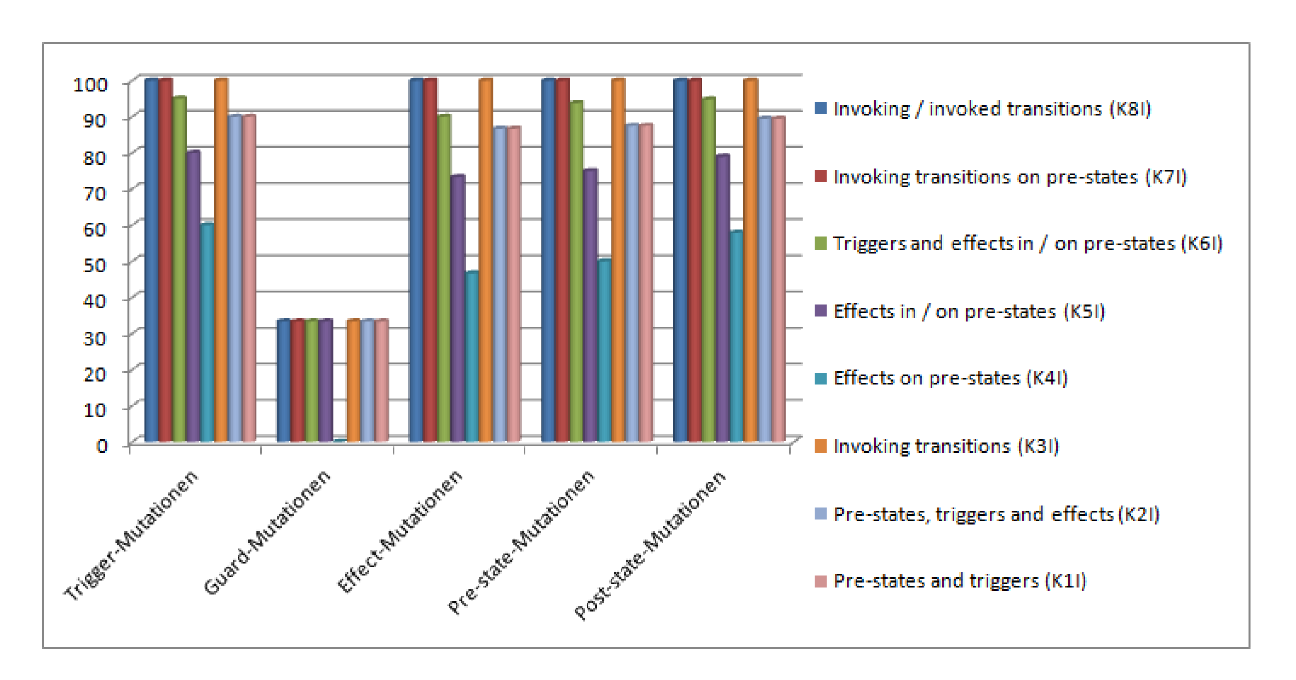


Abbildung 7.8.: Fehlererkennungspotential bzgl. einzelner Mutationsklassen von Testfallmengen, die zustandsbasierte Integrationstestkriterien auf dem Modell Infotainment erfüllen

Abbildung 7.9 zeigt die detaillierten Ergebnisse bezüglich der einzelnen Mutationsklassen auf dem Modell Liegenprüfplatz. Diese Ergebnisse wurden bereits in [PS10] veröffentlicht. Die Testfallmenge K8L entdeckt nur geringfügig mehr Mutanten als K7L. Hier wird ersichtlich, wo dieser kleine Unterschied entsteht, und zwar bezüglich der Effect-Mutationen. Sonst schneiden diese zwei Testfallmengen bezüglich der Klassen Trigger-, Effect-, Pre-state- und Post-state-Mutationen sehr gut (jeweils über 90%) ab. Ähnlich wie im Falle der Testfallmengen, die anhand des Infotainment-Modells generiert wurden, ist die Erkennbarkeit bzgl. Guard-Mutationen schlecht (nur 44,64%).

Zusammenfassend kann gesagt werden, dass bezüglich vier der fünf Mutationsklassen Testfälle, die dem anspruchsvollsten zustandsbasierten Kriterium genügen, sehr gut abschneiden. Dies bedeutet, dass Fehler, die sich auf den Trigger, Effect, Pre-state und Post-state von Transitionen

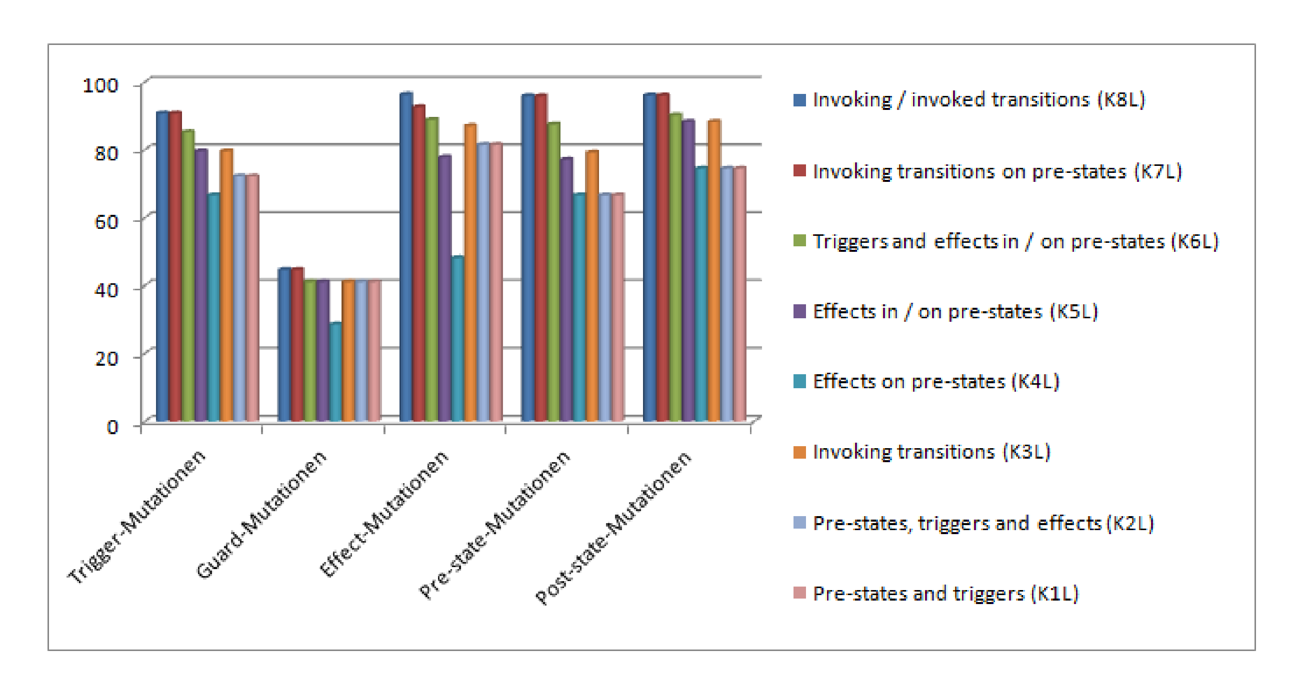


Abbildung 7.9.: Fehlererkennungspotential bzgl. einzelner Mutationsklassen von Testfallmengen, die zustandsbasierte Integrationstestkriterien auf dem Modell Liegenprüfplatz erfüllen

beziehen, die zu Mappings gehören, mit einer hohen Wahrscheinlichkeit entdeckt werden. Zur Erkennung von Guard-Fehlern eignen sich die zustandsbasierten Integrationstestkriterien allerdings nicht. Dies wurde auch für Fehlerarten beobachtet, die durch Komponententestfälle entdeckt werden. Die Erklärung dafür aus Abschnitt 7.1.3.1 lässt sich auch auf Integrationstestfälle übertragen.

Anhand der Abbildungen 7.7, 7.8 und 7.9 ist auch zu erkennen, dass die Integrationstestfallmengen, die das anspruchsvollste Integrationstestkriterium erfüllen, bezüglich der Effect-Mutationen sehr gute Ergebnisse liefern. Dies weicht von den Beobachtungen auf Komponentenebene[20] ab. Das liegt daran, dass die hier betrachteten Effect-Mutationen von Modell-Mutationsoperatoren erzeugt wurden, die Aufrufe betrachten. Da die für den Integrationstest generierten Testfälle speziell auf die Überdeckung solcher Aufrufe zwischen Komponenten zielen, wird diese Art von Mutationen auch zu einem hohen Anteil erkannt.

[20]Siehe 7.1.3.1

7.2. Erkennungspotential hinsichtlich Implementierungsfehler

Nachdem der vorherige Abschnitt die Möglichkeit der Erkennung von Modellfehlern betrachtet, widmet sich dieser Abschnitt der Untersuchung, ob mit modellbasiert generierten Testfällen Implementierungsfehler entdeckt werden können. Diese Evaluierung wurde anhand der Liegenprüfplatz-Anwendung durchgeführt, die in Abschnitt 6.1 beschrieben ist. Für diese Anwendung liegt neben dem UML-Modell auch eine manuell erstellte textuelle Testfallspezifikation vor. Anhand des UML-Modells wurden Testfälle generiert und untersucht, inwiefern diese die Erkennung von Implementierungsfehlern ermöglichen. Zusätzlich dazu wurde der Prozess der manuellen Testfallerstellung mit der modellbasierten Generierung verglichen und in Tabelle 7.3 beschrieben. Zusammengefasst wurden diese Ergebnisse bereits in [PS10] vorgestellt.

	TC	TS	EÜ	ME	DTS
Modellbasierte Generierung	36	504	100%	18 Stunden	2 Minuten
Manuelle Erstellung	2	117	58%	12 Stunden	6 Minuten

TC: Anzahl der Testfälle
TS: Anzahl der Teststeps
EÜ: Erreichte Überdeckung
ME: Manuelle Erstellungsdauer (Modellerstellung vs. Testfallerstellung)
DTS: Durchschnittliche Dauer pro Teststep

Tabelle 7.3.: Vergleich des Aufwands: automatische, modellbasierte Generierung (Kriterium: *Invoking / invoked transitions*) vs. manuelle Erstellung

Die Spalte *ME* beschreibt den manuellen Aufwand, der in beiden Fällen entstanden ist. Die manuell erstellten Testfälle wurden anhand der Spezifikation erstellt und in ein Text-Dokument zusammengefasst. Der Aufwand für die Erstellung der Testfälle betrug ca. 12 Stunden. Dabei wurden 2 sehr lange Testfälle generiert, die insgesamt 171 Teststeps enthalten. Für die Evaluierung des Potentials modellbasiert generierter Testfälle bzgl. der Erkennung von Implementierungsfehlern wurde die in Abschnitt 6.4 beschriebene Testfallmenge, die das *Invoking / invoked transitions* Kriterium erfüllt, betrachtet. Da diese Testfallmenge automatisch generiert wurde, entstand kein manueller Aufwand bei deren Erstellung. Allerdings benötigte die Erstellung des der Testfallgenerierung zugrunde liegenden Modells ca. 18 Stunden.

Bei Betrachtung der Anzahl enthaltener Teststeps fällt auf, dass die automatisch generierte Testfallmenge mehr als viermal mehr Teststeps enthält, als die manuell erstellte Testfallmenge. Die dabei resultierende Überdeckung beträgt im Falle der automatisch generierten Testfallmenge 100% bezüglich der ausführbaren Mappings. Dagegen beträgt die von der manuell erzeugten Testfallmenge auf dem Modell erzielte Überdeckung (bezüglich aller ausführbaren Mappings) bloß 58%. Sowohl hinsichtlich Umfang als auch Modellüberdeckung sind diese zwei Testfallmengen also sehr unterschiedlich. Insofern lohnt sich das aufwändigere modellbasierte Verfahren erst dann, wenn dadurch Fehler entdeckt werden, die von den manuell erstellten Testfällen nicht erkannt werden.

Durch Ausführung der automatisch generierten Testsuite, wurden drei Fehler entdeckt, die von der manuell erzeugten Testsuite nicht entdeckt wurden. Bei den dabei aufgefundenen Fehlern handelte es sich um Fehler, die sich bei einer vom normalen Nutzungsprofil abweichenden Benutzung der Software manifestierten. Das normale Nutzungsszenario sieht dabei wie folgt aus: es werden sequentiell Prüfschritte durchgeführt. Jeder Prüfschritt wird vom Benutzer gestartet; nach der Durchführung des Schrittes wird das Ergebnis angenommen oder abgelehnt, um danach den nächsten Prüfschritt zu betrachten. Bei dieser rein sequentiellen Durchführung der Schritte ergaben sich keine Probleme. Fehlerhaftes Verhalten ergab sich erst bei einer davon abweichenden Benutzung, z. B. wenn ein bestimmter Prüfschritt während der Durchführung abgebrochen wurde.

Zwei der drei Implementierungsfehler führten dazu, dass in solchen Fällen der Status der Prüfschritte nicht richtig abgespeichert wurde. So muss etwa nach dem Abbruch eines Prüfschrittes dessen Status auf „nicht ausgeführt“ zurückgesetzt werden, was allerdings in der implementierten Anwendung nicht passierte, hier wurde der Status auf „nicht bestanden“ gesetzt. Diese zwei Fehler waren auf die inkorrekte Interaktion der Steuerungskomponente Achstest mit der Komponente PartialResult zurückzuführen. Der dritte Fehler offenbarte sich beim Wechseln zwischen Prüfschritten, während ein bestimmter Prüfschritt gerade abgearbeitet wurde. In diesem Fall sollte ein Bestätigungsdialog aufkommen. Da dieser Dialog nicht angezeigt wurde, wich das implementierte Verhalten allerdings davon ab. Hier handelte es sich also um einen Fehler in der Steuerungskomponente.

Zusammenfassend kann gesagt werden, dass die entdeckten Fehler daraus resultieren, dass die im Modell beschriebene Funktionalität im Rahmen der Implementierungsphase nicht richtig umgesetzt wurde. Durch manuell erstellte Testfälle wurden diese Fehler nicht erkannt, da bei

der manuellen Erstellung der Testfälle Ausnahmesituationen nicht betrachtet wurden. Für die Erkennung solcher Fehler eignet sich die systematische, modellbasierte Vorgehensweise. Darüber hinaus, besteht ein weiterer großer Vorteil der modellbasierten Generierung von Testfällen darin, dass die Wartung der Testfälle (im Falle von Modelländerungen) bedeutend vereinfacht wird.

8. Zusammenfassung und Ausblick

In Rahmen dieser Arbeit wurde ein werkzeuggestütztes Verfahren vorgestellt, das die automatische Testfallgenerierung anhand von UML-Zustandsautomaten ermöglicht. Zu den unterstützten modellbasierten Überdeckungskriterien zählen sowohl Komponententestkriterien als auch Integrationstestkriterien. Was die unterstützten Integrationstestkriterien betrifft, knüpft diese Arbeit an die Publikation [SOP07] an, in der mehrere modellbasierte Integrationstestkriterien eingeführt wurden.

Das Verfahren wurde im Rahmen des Verbundprojekts UnITeD (an dem die Kooperationspartner Lehrstuhl für Software Engineering und Afra GmbH mitwirkten) entwickelt. Bei der Durchführung des Projekts konnte auf die am Lehrstuhl bereits vorhandene Erfahrung mit dem praktischen Einsatz genetischer Algorithmen zurückgegriffen werden. Einen wichtigen Beitrag zu diesem Projekt leisteten zwei Abschlussarbeiten: in [Sch06] wurde ein Modellsimulator für Zustandsautomaten entwickelt; im Rahmen der Arbeit von [Neu08] wurde das Verfahren zur Visualisierung der Modellüberdeckung konzipiert und das benötigte Plugin für MagicDraw umgesetzt.

Das implementierte Software-Werkzeug erlaubt Komponenten und Integrationstestfälle automatisch zu generieren und hinsichtlich zweier Ziele zu optimieren: einerseits hinsichtlich einer hohen Überdeckung des Modells und anderseits hinsichtlich einer geringen Anzahl an Testfällen. Praktisch wurde das Verfahren anhand mehrerer Modelle evaluiert; die angestrebte Überdeckung konnte, bis auf eine Ausnahme, in allen Fällen erreicht werden.

Des Weiteren wurde im Rahmen dieser Arbeit der Frage nach dem Fehlererkennungspotential von Testfallmengen nachgegangen, die modellbasierte Überdeckungskriterien für den Komponententest oder Integrationstest erfüllen. Um zu untersuchen, ob mit Hilfe modellbasierter Testfälle Modellierungsfehler entdeckt werden können, wurde ein modellbasiertes Mutationstestverfahren von Grund auf neu entwickelt. Für diesen Zweck mussten zunächst Fehlerarten bei der Modellierung von Zustandsautomaten identifiziert und kategorisiert werden. Darauf aufbau-

end, wurden entsprechende Modell-Mutationsoperatoren implementiert. Die Mutationsanalyse ermöglicht sowohl die Bestimmung des gesamten Anteils an erkannten Mutanten als auch die Bestimmung der Ergebnisse bezüglich einzelner Mutationsklassen.

Es stellte sich dabei heraus, dass die Testfallmengen, die den anspruchsvollsten Kriterien auf Komponenten- oder Integrationsebene genügen, bezüglich der meisten Arten von Mutanten gute bis sehr gute Ergebnisse erreichten. Somit wurde gezeigt, dass sich modellbasierte Testfälle zur Erkennung von Fehlern in Zustandsautomaten eignen. Wie man leicht erklären konnte, wurden vor allem Fehler in Guards und Variablenzuweisungen nicht zu einem befriedigenden Anteil erkannt.

Auch die Möglichkeit der Erkennung von Implementierungsfehlern durch modellbasierte Testfälle wurde untersucht und mit der Fehlererkennung von manuell erstellten Testfallmengen verglichen. So verursacht die systematische Vorgehensweise anhand von Modellen zwar einen höheren Aufwand; dieser lohnt sich allerdings, da mit Hilfe modellbasierter Testfälle Implementierungsfehler gefunden wurden, die durch die manuell erzeugten Testfälle nicht entdeckt wurden.

Nachdem der Beitrag der Arbeit zusammengefasst wurde, folgt nun ein kurzer Ausblick. Das im Rahmen dieser Arbeit vorgestellte Verfahren unterstützt den Test von Softwaresystemen, die aus *synchron kommunizierenden Komponenten* zusammengesetzt sind. Eine immer wichtiger werdende Klasse von Systemen, und zwar die so genannten *autonomen Systeme*, waren dagegen nicht im Fokus dieser Arbeit. Solche Systeme kommen z. B. in der industriellen Produktion vor und zeichnen sich meistens durch eine dezentralisierte Architektur aus. Das führt dazu, dass die einzelnen Subsysteme weitgehend autonom arbeiten, gleichzeitig aber um den Zugriff auf gemeinsame Ressourcen konkurrieren.

Damit autonome Systeme verlässlich funktionieren ist es unerlässlich, einerseits *Modellierungssprachen zu ermitteln*, die es ermöglichen, die Vielfalt der Interaktionsszenarien zu erfassen und andererseits *Testverfahren zu entwickeln*, die zur Überprüfung solcher Systeme geeignet sind. Im Rahmen des europäisches Verbundprojekts *Robust & Safe Mobile Co-operative Autonomous Systems (R3-Cop)*, das sich mit dem Entwurf robuster und sicherer autonomer Systeme für sicherheitskritische Bereiche beschäftigt, werden unter anderem diese zwei Zielsetzungen verfolgt. Bisher wurden am Lehrstuhl für Software Engineering in Erlangen geeignete Notationen für die Beschreibung solcher Systeme betrachtet und in [SSL11] vorgestellt. In einem weiteren Schritt sollen nun geeignete Verfahren für den Test solcher Systeme entwickelt werden, was zunächst voraussetzt, dass adäquate Überdeckungskriterien für die identifizierten Modellnotationen

definiert werden. Dass genetische Algorithmen bereits erfolgreich für die Testfallgenerierung anhand von Java-Code (in der Arbeit [Ost07]) und von UML-Modellen (im Rahmen dieser Arbeit) eingesetzt werden konnten, lässt vermuten, dass sie auch für die Testfallgenerierung im Kontext autonomer Systeme geeignet sind. Diese Fragestellung ist Gegenstand aktueller Untersuchungen.

Literaturverzeichnis

[ABLN06] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. Technical report, Carleton University, 2006.

[ABuR+06] Shaukat Ali, Lionel C. Briand, Muhammad Jaffar ur Rehman, Hajra Asghar, Muhammad Zohaib Z. Iqbal, and Aamer Nadeem. A State-based Approach to Integration Testing based on UML Models. Technical report, 2006.

[AFGC03] Anneliese Andrews, Robert France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13, 2003.

[AO00] Aynur Abdurazik and Jeff Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. Proc. of 3rd International Conference on The Unified Modeling Language:13, 2000.

[AO04] R. T. Alexander and J. Offutt. Coupling-based Testing of O-O Programs. *Journal of Universal Computer Science*, 10(4):391–427, 2004.

[Bal97] Helmut Balzert. *Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung und Unternehmensmodellierung*. Spektrum-Akademischer Vlg, 1997.

[BB00] F. Basanieri and Antonia Bertolino. A practical approach to UML-based derivation of integration test. In *Proceedings of 4th International Quality Week Europe*, 2000.

[BDG+04] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Serge Lucio, Eric Samuelsson, Ina Schieferdecker, and Clay E. Williams. The UML 2.0 Testing Profile. In *Conquest 2004*, 2004.

[BH08] Fevzi Belli and Axel Hollmann. Test Generation and Minimization with “Basic“

Statecharts. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 718–723, New York, NY, USA, 2008. ACM.

[BLS02] L. C. Briand, Y. Labiche, and G. Soccar. Automating Impact Analysis and Regression Test Selection Based on UML Designs. In *Proceedings of the International Conference on Software Maintenance (ICSM02)*, 2002.

[BRJ06] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Das UML-Benutzerhandbuch*. Addison-Wesley, München [u.a.], 2006.

[Cho78] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.

[CNM07] E.G. Cartaxo, F.G.O. Neto, and P.D.L. Machado. Test case generation by means of UML sequence diagrams and labeled transition systems. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 1292 –1297, 2007.

[CPU07] Yanping Chen, Robert L. Probert, and Hasan Ural. Model-based Regression Test Suite Generation Using Dependence Analysis. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 54–62, New York, NY, USA, 2007. ACM.

[Dar59] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859.

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. Technical report, Technological University Eindhoven, 1970.

[DMM01] Márcio E. Delamaro, José C. Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Trans. Softw. Eng.*, 27(3):228–247, 2001.

[DS06] Wolfgang Domschke and Armin Scholl. Heuristische verfahren. Jenaer Schriften zur Wirtschaftswissenschaft 08/2006, Friedrich-Schiller-Universität Jena, Wirtschaftswissenschaftliche Fakultät, 2006.

[DT07] T. Dinh-Trong. *A Systematic Approach to Testing UML Designs*. Dissertation, Colorado State University, Fort Collins, Colorado, 2007.

[FIMN07] Q. Farooq, M. Iqbal, Z. Malik, and A. Nadeem. An Approach for Selective State Machine based Regression Testing. In *Proceedings of the 3rd international work-*

shop on Advances in model-based testing, pages 44–52, New York, NY, USA, 2007. ACM.

[FMSM99] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. Mutation Testing Applied to Validate Specifications Based on Statecharts. In *ISSRE '99: Proceedings of the 10th International Symposium on Software Reliability Engineering*, page 210, Washington, DC, USA, 1999. IEEE Computer Society.

[GVEB08] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Intl Thomson Business Pr, 2008.

[Hof08] Dirk W. Hoffmann. *Software-Qualität*. Springer, 2008.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[IEE90] IEEStd.610.12-1990. *IEE Std. 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[IEE91] IEEEStd.610-1991. *IEEE Std. 610-1991: IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[Ist10] Standard glossary of terms used in Software Testing, 2010.

[JH09] Yue Jia and Mark Harman. Higher Order Mutation Testing. *Inf. Softw. Technol.*, 51:1379–1393, October 2009.

[JO98] Zhenyi Jin and A. Jefferson Offutt. Coupling-based Criteria for Integration Testing. *The Journal of Software Testing, Verification, and Reliability*, 8:133–154, 1998.

[JS05] Martin Jung and Francesca Saglietti. Supporting Component and Architectural Reusage by Detection and Tolerance of Integration Faults. In *9th IEEE International Symposium on High Assurance Systems Engineering (HASE '05)*. IEEE Computer Society, 2005.

[Jun07] Martin Jung. *Modellbasierte Generierung von Beherrschungsmechanismen für Inkonsistenzen in komponentenbasierten Systemen*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2007.

[KHC+99] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test Cases Generation

from UML State Diagrams. In *In IEE Proceedings: Software*, pages 187–192, 1999.

[KR03] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-Generating Test Case Using UML Statechart Diagrams. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 296–300, , Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.

[KTS10] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software Test Automation in Practice: Empirical Observations. *Advances in Software Engineering*, 2010, January 2010.

[LI07] R. Lefticaru and F. Ipate. Automatic State-Based Test Generation Using Genetic Algorithms. In *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*, 2007.

[Lig09] Peter Liggesmeyer. *Software-Qualität. Testen, Analysieren und Verifizieren von Software.* 2009.

[Lyu96] Michael R. Lyu. *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[MS11] Matthias Meitner and Francesca Saglietti. Software Reliability Assessment based on Operational Representativeness and Interaction Coverage. In *24th International Conference on Architecture of Computing Systems (ARCS 2011)*, 2011.

[Neu08] Achim Neubauer. Visualisierung überdeckter sowie zu überdeckender Modellelemente im modellbasierten Test. Diplomarbeit, Lehrstuhl für Software Engineering, Universität Erlangen-Nürnberg, 2008.

[NR07] Leila Naslavsky and Debra J. Richardson. Using Traceability to Support Model-Based Regression Testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 567–570, New York, NY, USA, 2007. ACM.

[NSVT07] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empiri-*

cal assessment of software engineering languages and technologies, pages 31–36, New York, NY, USA, 2007. ACM.

[OMG10] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, 2010.

[OSSP07] Norbert Oster, Claudia Schieber, Francesca Saglietti, and Florin Pinte. Automatische, modellbasierte Testdatengenerierung durch Einsatz evolutionärer Verfahren. In Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, and Marc Ronthaler, editors, *Informatik 2007 - Informatik trifft Logistik (Band 2)*, volume P-110 of *Lecture Notes in Informatics*, pages 398–403, Bonn, 2007. Köllen Druck+Verlag GmbH.

[Ost07] Norbert Oster. *Automatische Generierung optimaler struktureller Testdaten für objekt-orientierte Software mittels multi-objektiver Metaheuristiken*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, 2007.

[PBSO08] Florin Pinte, Gerhard Baier, Francesca Saglietti, and Norbert Oster. Automatische Generierung optimaler modellbasierter Regressionstests. In Heinz-Gerd-Hegering, Axel Lehmann, Hans-Jürgen Ohlbach, and Christian Scheideler, editors, *Informatik 2008 - Beherrschbare Systeme dank Informatik (Band 1)*, volume P-133 of *Lecture Notes in Informatics (LNI)*, pages 193–198, Bonn, September 2008. Gesellschaft für Informatik e.V. (GI), Köllen Druck+Verlag GmbH.

[POS08] Florin Pinte, Norbert Oster, and Francesca Saglietti. Techniques and tools for the automatic generation of optimal test data at code, model and interface level. In *ICSE Companion '08: Companion of the 13th International Conference on Software engineering (ICSE 2008)*, pages 927–928, New York, NY, USA, 2008. ACM.

[PS10] Florin Pinte and Francesca Saglietti. Evaluierung des Fehlererkennungspotentials modellbasierter Komponenten- und Integrationstestfälle. In Klaus-Peter Fähnrich and Bogdan Franczyk, editors, *Informatik 2010 - Service Science – Neue Perspektiven für die Informatik (Band 2)*, volume P-176 of *Lecture Notes in Informatics*, pages 375–380, Bonn, 2010. Köllen Druck+Verlag GmbH.

[PSN09] Florin Pinte, Francesca Saglietti, and Achim Neubauer. Visualisierung überdeckter sowie zu überdeckender Modellelemente im modellbasierten Test. In Stefan Fischer, Eric Maehle, and Rüdiger Reischuk, editors, *Informatik 2009 - Im Fokus das Leben*, volume P-154 of *Lecture Notes in Informatics*, page 358, Bonn, 2009. Köllen Druck+Verlag GmbH.

[PSO08] Florin Pinte, Francesca Saglietti, and Norbert Oster. Automatic Generation of Optimized Integration Test Data by Genetic Algorithms. In Walid Maalej and Bernd Bruegge, editors, *Software Engineering 2008 - Workshopband*, volume P-122 of *Lecture Notes in Informatics (LNI)*, pages 415–422. Gesellschaft für Informatik, February 2008. Workshop proceedings of the International Conference on Software Engineering 2008 (SE 2008).

[PUA06] O. Pilskalns, G. Uyan, and A. Andrews. Regression Testing UML Designs. In *Proc. of 22nd IEEE International Conference on Software Maintenance*, pages 254–264, Sept. 2006.

[RBGW10] T. Roßner, C. Brandes, H. Götz, and M. Winter. *Basiswissen Modellbasierter Test*. dpunkt.verlag, 2010.

[RHQ+07] Chris Rupp, Jürgen Hahn, Stefan Queins, Mario Jeckle, and Barbara Zengler. *UML 2 glasklar. Praxiswissen für die UML -Modellierung und Zertifizierung*. Hanser Fachbuchverlag; Auflage: 2., 2007.

[RJBP05] M. Rehman, F. Jabeen, A. Bertolino, and A. Polini. Software Component Integration Testing: A Survey. Technical Report 2005-TR-41, Italian National Research Council, 2005.

[RKS05] Atanas Rountev, Scott Kagan, and Jason Sawin. Coverage Criteria for Testing of Object Interactions in Sequence Diagrams. *In Fundamental Approaches to Software Engineering, LNCS 3442*, pages 282–297, 2005.

[RW85] S. Rapps and E.J. Weyuker. Selecting Software Test Data Using Data Flow Information. *Software Engineering, IEEE Transactions on*, SE-11(4):367 – 375, 1985.

[Sch05] Peter Scholz. *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*. Springer, Berlin, 2005.

[Sch06] Dominik Schindler. Bewertender Vergleich und Erweiterung unterschiedlicher UML-Simulatoren zur Bestimmung der Modellüberdeckung. Diplomarbeit, Lehrstuhl für Software Engineering, Universität Erlangen-Nürnberg, 2006.

[Söh10] Sven Söhnlein. *Quantitative Bewertung der Softwarezuverlässigkeit komponentenbasierter Systeme durch statistische Auswertung der Betriebserfahrung*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2010.

[SJ04] Francesca Saglietti and Martin Jung. Classification, Analysis and Detection of Interface Inconsistencies in Safety-Relevant Component-based Systems. In U. Schmocker C. Spitzer and V. N. Dang, editors, *Proceedings of Probabilistic Safety Assessment and Management*, volume 4, pages 1864 – 1869. Springer-Verlag, 2004.

[SM10] Santosh Kumar Swain and Durga Prasad Mohapatra. Test Case Generation from Behavioral UML Models. *International Journal of Computer Applications*, 6, 2010.

[SMFS00] Simone Do Rocio Senger De Souza, José Carlos Maldonado, Sandra Camargo Pinto Ferraz Fabbri, and Wanderley Lopes De Souza. Mutation Testing Applied to Estelle Specifications. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, 2000.

[Sok04] Dehla Sokenou. Testfallgenerierung aus Statecharts und Interaktionsdiagrammen. *Softwaretechnik-Trends*, 24, 2004.

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2010.

[SOP07] Francesca Saglietti, Norbert Oster, and Florin Pinte. Interface Coverage Criteria Supporting Model-Based Integration Testing. In Marco Platzner, Karl-Erwin Großpietsch, Christian Hochberger, and Andreas Koch, editors, *ARCS '07 - Workshop Proceedings, Workshop proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS 2007)*, pages 85–93, Zürich, 2007. VDE Verlag GmbH Berlin/Offenbach.

[SOP08] Francesca Saglietti, Norbert Oster, and Florin Pinte. White and Grey-Box Verification and Validation Approaches for Safety- and Security-Critical Software Systems. *Information Security Technical Report*, 13(1):10–16, March 2008. DOI: 10.1016/j.istr.2008.03.002.

[SP10] Francesca Saglietti and Florin Pinte. Automated Unit and Integration Testing for Component-based Software Systems. In Brahim Hamid, Carsten Rudolph, and Christoph Ruland, editors, *International workshop on Security and Dependability for Resource Constrained Embedded Systems*, 2010.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[SPS09] Francesca Saglietti, Florin Pinte, and Sven Söhnlein. Integration and Reliability Testing for Component-based Software Systems. In *Proc. of 35th EUROMICRO SEAA 2009*, pages 368–374, 2009.

[SSL11] Francesca Saglietti, Sven Söhnlein, and Raimar Lill. Evolution of Verification Techniques by Increasing Autonomy of Cooperating Agents. *Studies in Computational Intelligence*, 2011.

[TP07] Jens Trompeter and Georg Pietrek, editors. *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Entwickler.Press, Frankfurt am Main, 2007.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.

[UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-based Testing. 2006. Department of Computer Science The University of Waikato Private Bag 3105 Hamilton, New Zealand.

[Utt05] Mark Utting. Position Paper: Model-Based Testing. *Verified Software: Theories, Tools, Experiments. ETH Zürich, IFIP WG*, 2, 2005.

[WCO03] Ye Wu, Mei-Hwa Chen, and Jeff Offutt. UML-based Integration Testing for Component-based Software. *Proc. Second International Conference on COTS-based Software Systems*, 2580, 2003.

[WS07] Stephan Weißleder and Bernd-Holger Schlingloff. Automatic Test Generation from Coupled UML Models using Input Partitions. In *Proc. of Int. Workshop on Model-Driven Engineering, Verification and Validation*, 2007.

[WS08] Stephan Weissleder and Bernd-Holger Schlingloff. Quality of Automatically Generated Test Cases based on OCL Expressions. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 517–520, Washington, DC, USA, 2008. IEEE Computer Society.

Abbildungsverzeichnis

Tabellenverzeichnis

A. UML Modelle

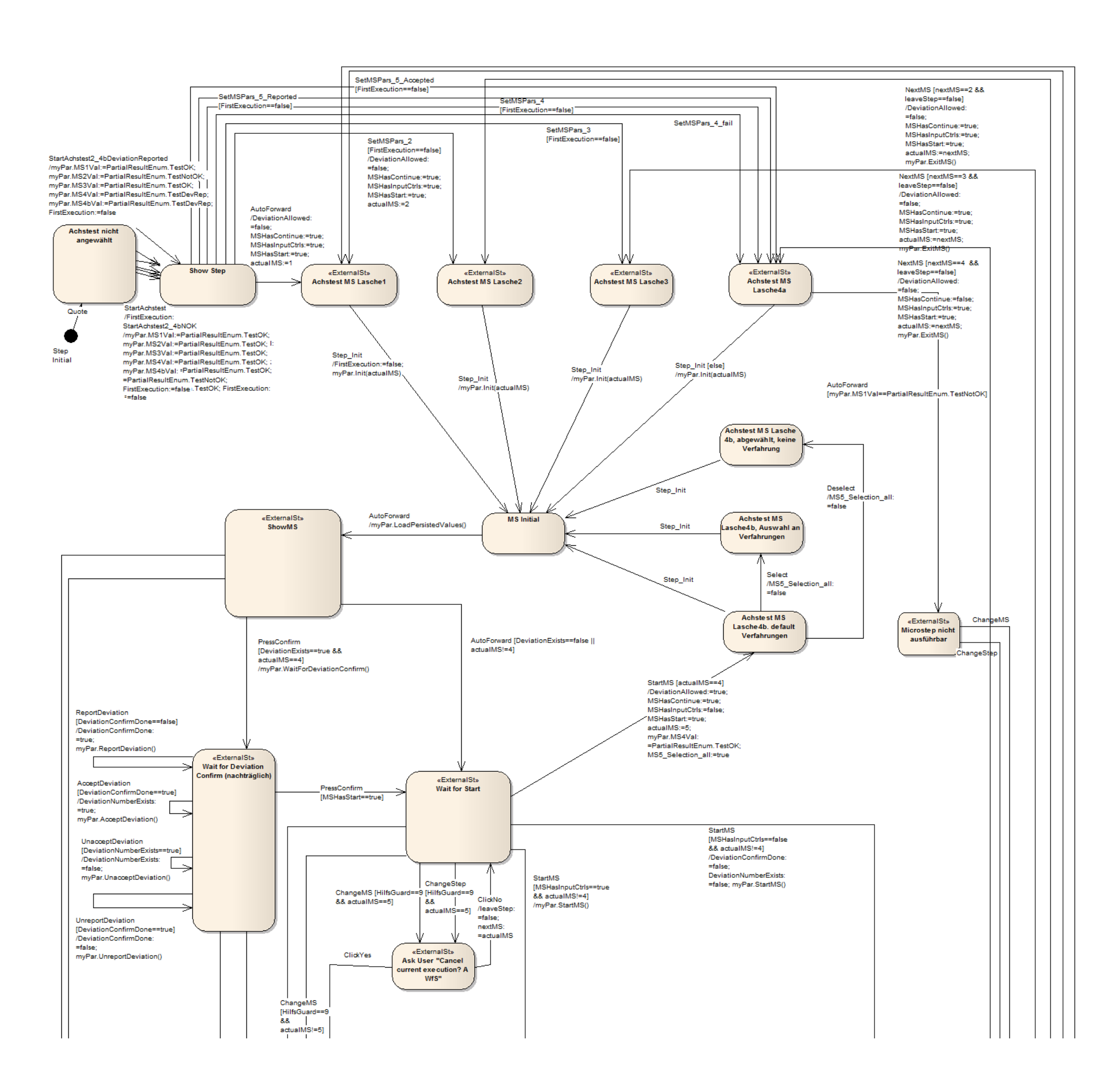


Abbildung A.1.: Modell Liegenprüfplatz, Komponente Achstest (Teil 1 von 2)

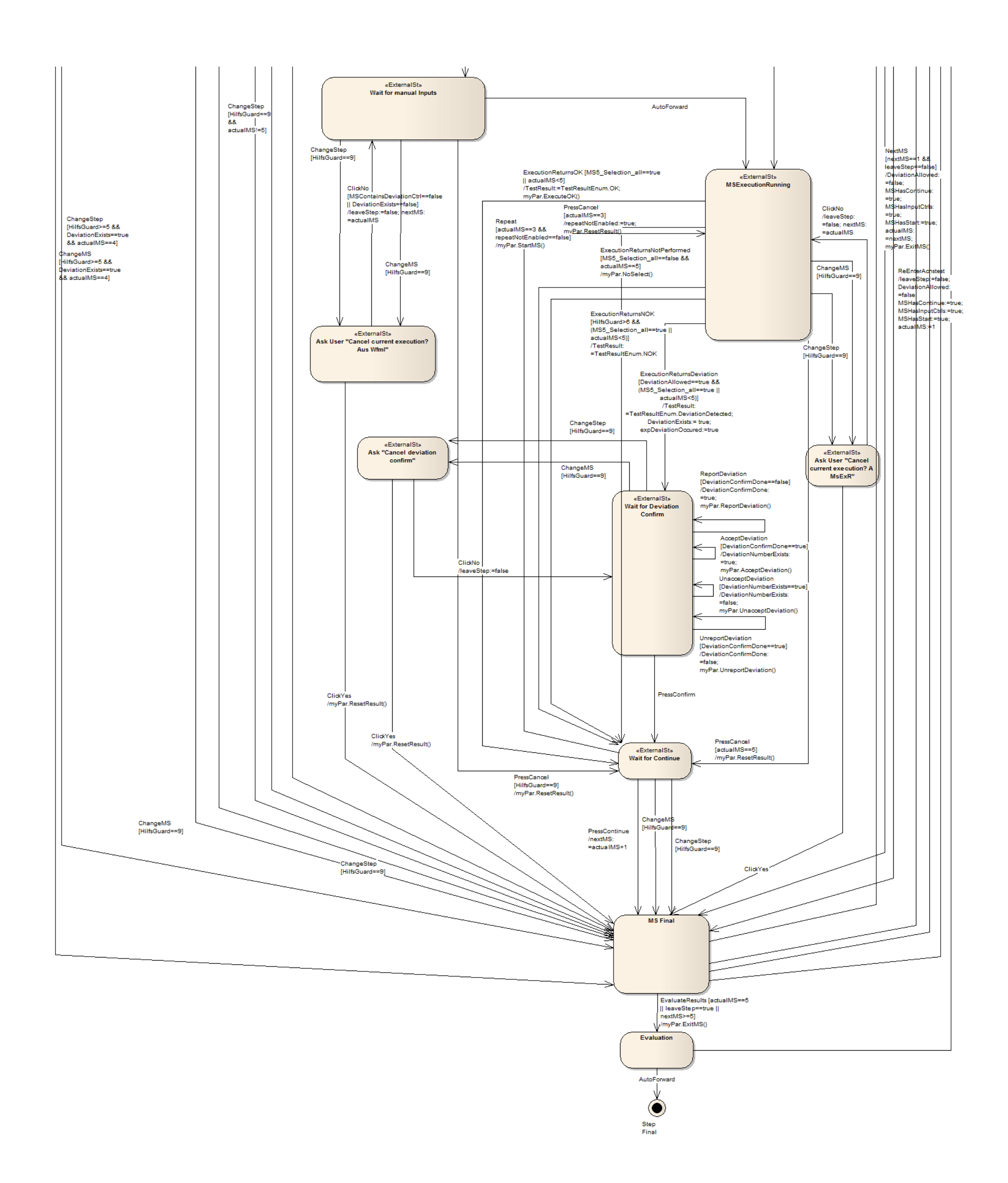


Abbildung A.2.: Modell Liegenprüfplatz, Komponente Achstest (Teil 2 von 2)

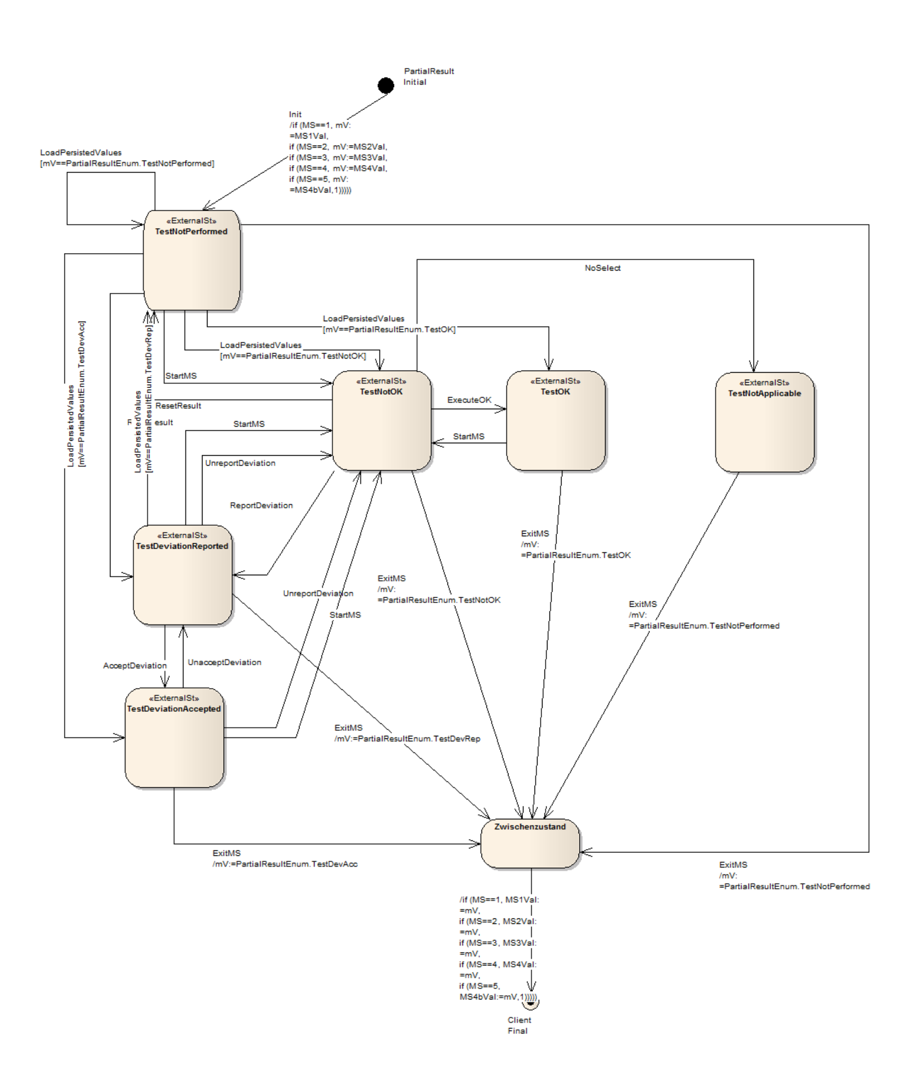


Abbildung A.3.: Modell Liegenprüfplatz, Komponente PartialResult

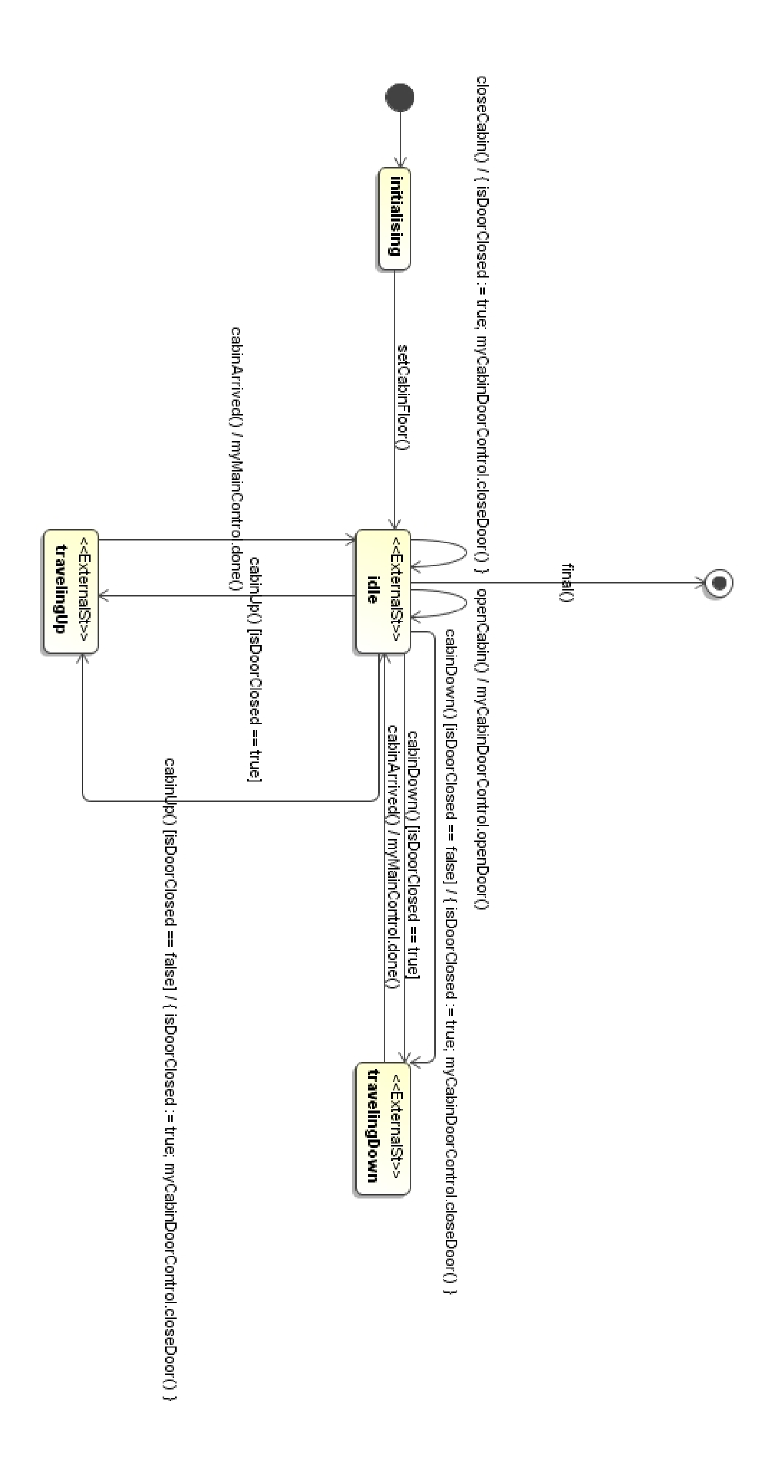


Abbildung A.4.: Modell CabinControl, Komponente CabinControl

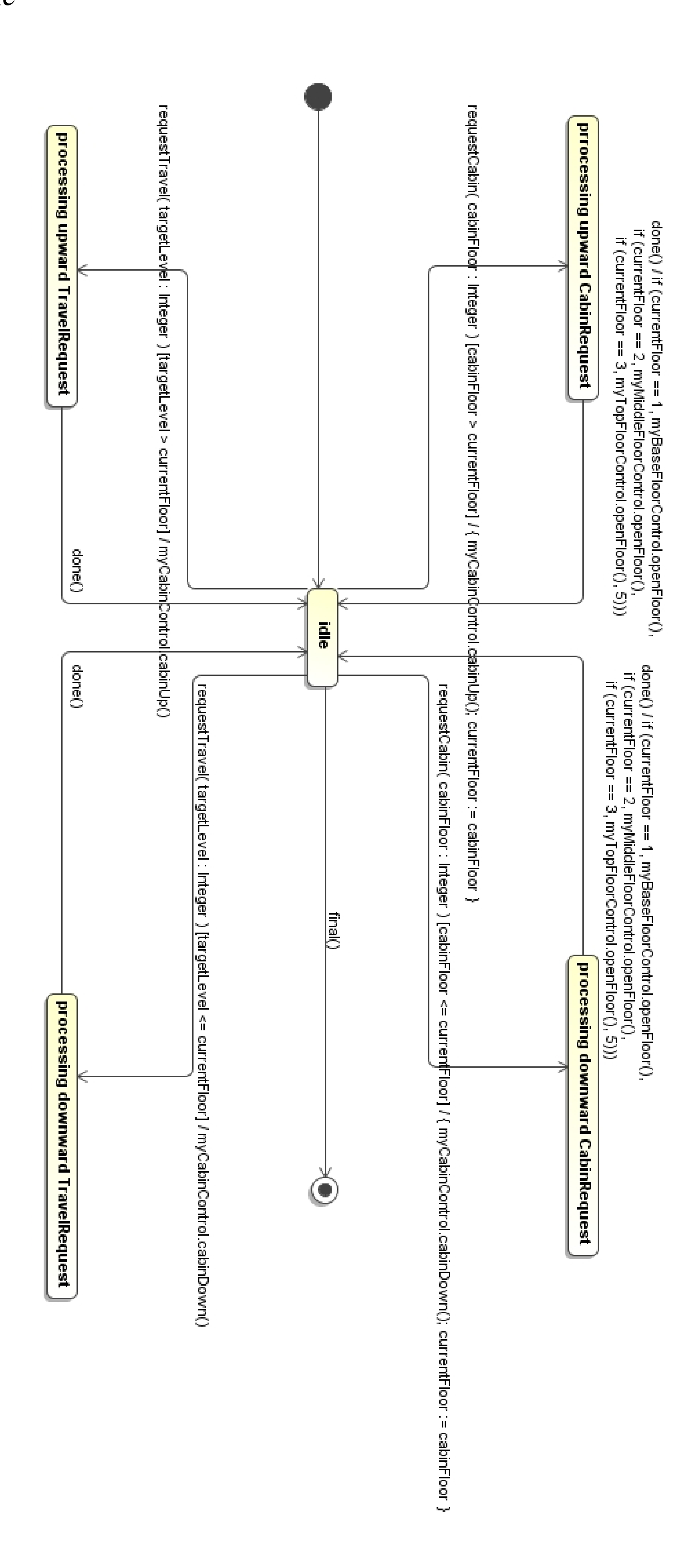


Abbildung A.5.: Modell CabinControl, Komponente MainControl

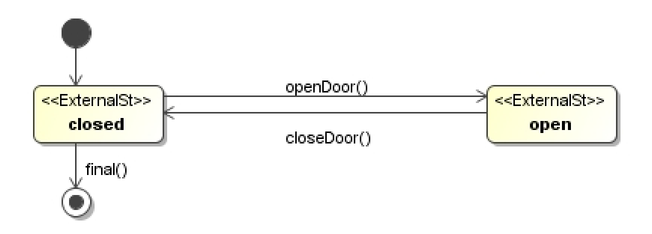


Abbildung A.6.: Modell CabinControl, Komponente CabinDoorControl

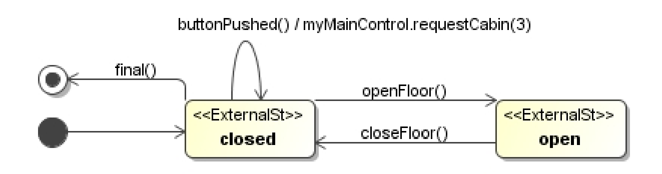


Abbildung A.7.: Modell CabinControl, Komponente TopFloorControl

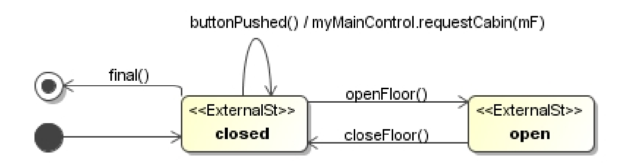


Abbildung A.8.: Modell CabinControl, Komponente MiddleFloorControl

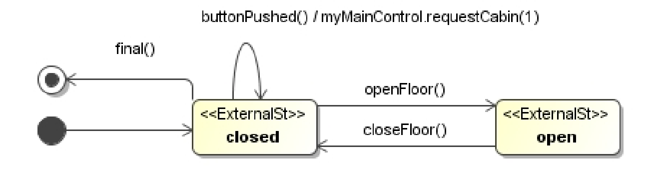


Abbildung A.9.: Modell CabinControl, Komponente BaseFloorControl

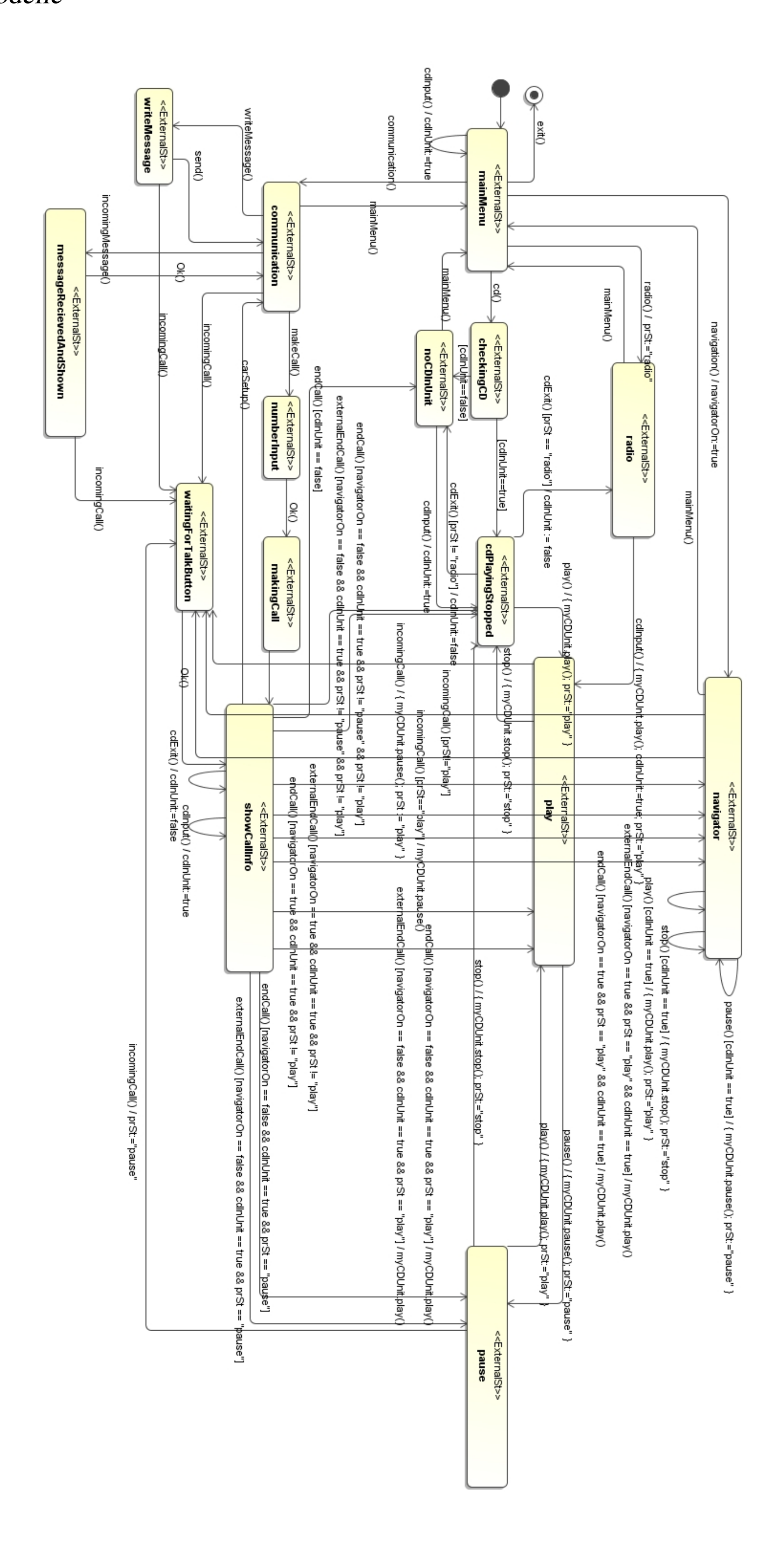


Abbildung A.10.: Modell Infotainment, Komponente InfotainmentGui

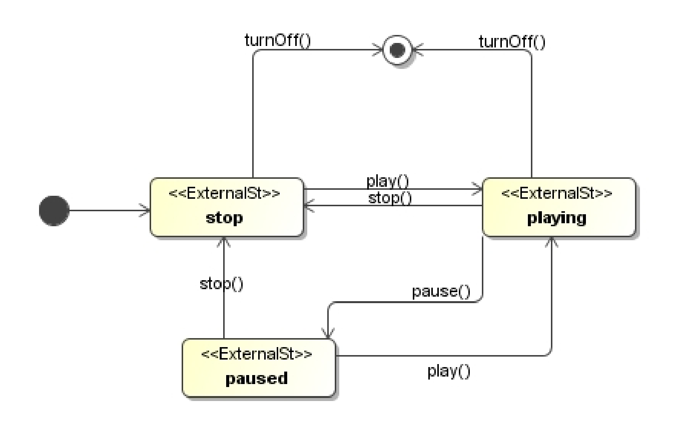


Abbildung A.11.: Modell Infotainment, Komponente CDPlayer

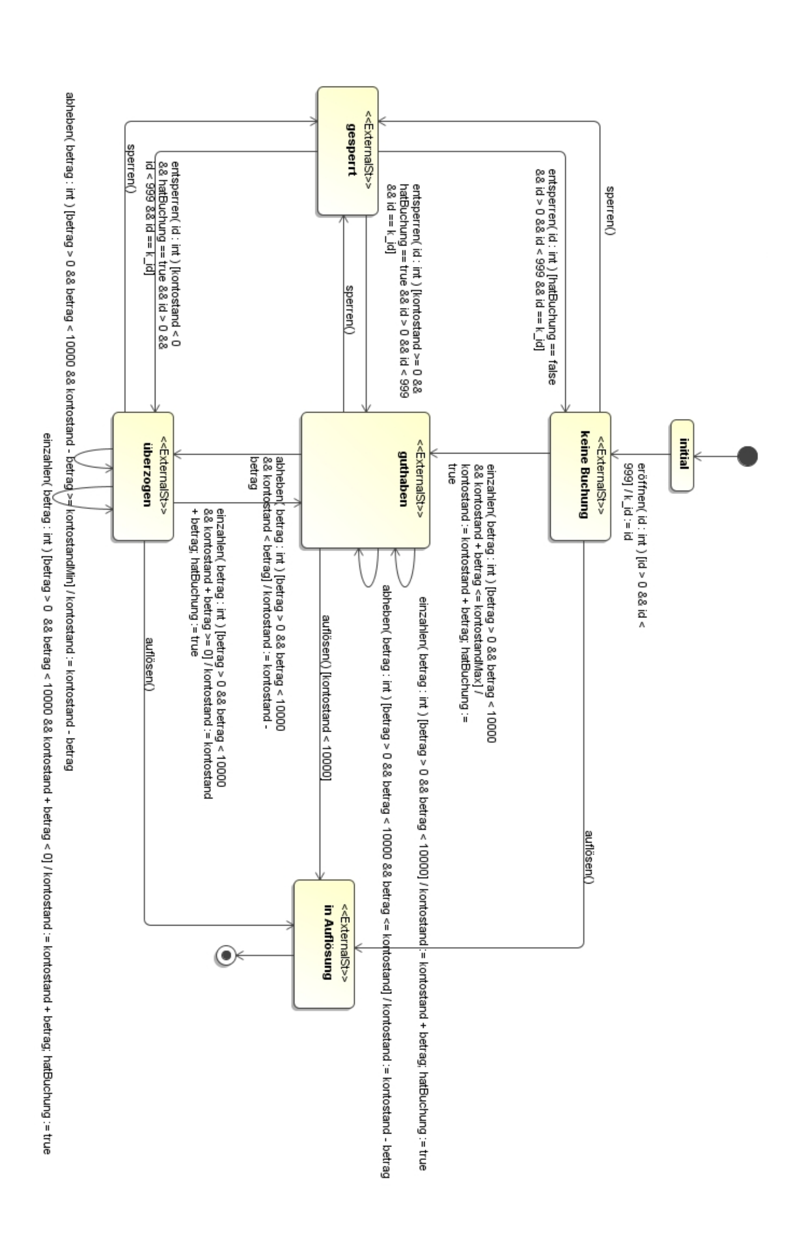


Abbildung A.12.: Modell Konto, Komponente Konto

B. Werkzeug UnITeD

Im Folgenden wird die Bedienung des Werkzeugs UnITeD beschrieben. Nach dem Start des Werkzeugs wird der Startbildschirm angezeigt (siehe B.1). Über den Menüpunkt *File* $\rightarrow$ *Load Model* (siehe B.2) kann eine Modelldatei geladen werden. Nach dem Laden wird das Modell im so genannten Modellbaum angezeigt (siehe B.3). Durch klicken auf einzelne Elemente des Baumes werden deren Eigenschaften im Tab *Properties* angezeigt.

Vor dem Start der Generierung kann der Konfigurationsdialog aufgerufen werden, der drei Tabs enthält. Im ersten Tab *Coverage Criterion* (siehe B.4) kann eingestellt werden, welche Kriterien verfolgt werden. Im zweiten Tab *Optimization Conditions* (siehe B.5) kann die Gewichtung für die Fitnessfunktion festgelegt werden. Im dritten Tab *Advanced* (siehe B.6) kann der genetische Algorithmus parametrisiert und das Abbruchkriterium ausgewählt werden.

Zum Starten der Testfallgenerierung muss der Tester einen TestContext auswählen und dann den *OK*-Button drücken (siehe B.7). Während der Testfallgenerierungsprozess läuft, wird der Fortschritt der Generierung angezeigt (siehe B.8). Der Prozess kann dabei jederzeit angehalten werden, indem der *Cancel Operation*-Button (unten rechts) angeklickt wird. Wenn dies geschieht, wird die bis dahin beste generierte Testsuite angezeigt.

Über das Kontextmenü der generierten Testsuite (siehe B.9) können weitere Funktionalitäten aufgerufen werden. So z. B. kann die Weiterführung des Generierungsvorgangs ausgehend von der bereits generierten Testsuite (*Continue Global Optimization*) oder das Speichern der Visualisierungsinformationen der Testsuite (*Save Visualisation Data*) ausgewählt werden.

Falls der Generierungsvorgang fortgesetzt wird, werden so lange Testfälle generiert, bis die ausgewählte Abbruchbedingung erfüllt ist oder die Generierung wieder mittels des *Cancel Operation*-Buttons unterbrochen wird. Dann wird die generierte Testsuite erneut angezeigt (siehe B.10). Durch anklicken eines bestimmten Testfalls wird dieser detailliert dargestellt (siehe B.11).

Neben der Veranschaulichung der generierten Testfälle können die Testfälle in verschiedene

Formate exportiert werden. Dies kann über den Menüpunkt *Testscripts* (siehe B.12) realisiert werden. Des Weiteren kann für eine Testsuite eine Mutationsanalyse durchgeführt werden. Dies wird über einen Dialog (siehe B.13) ermöglicht, der mittels des Menüpunkts *Mutation* aufgerufen wird.

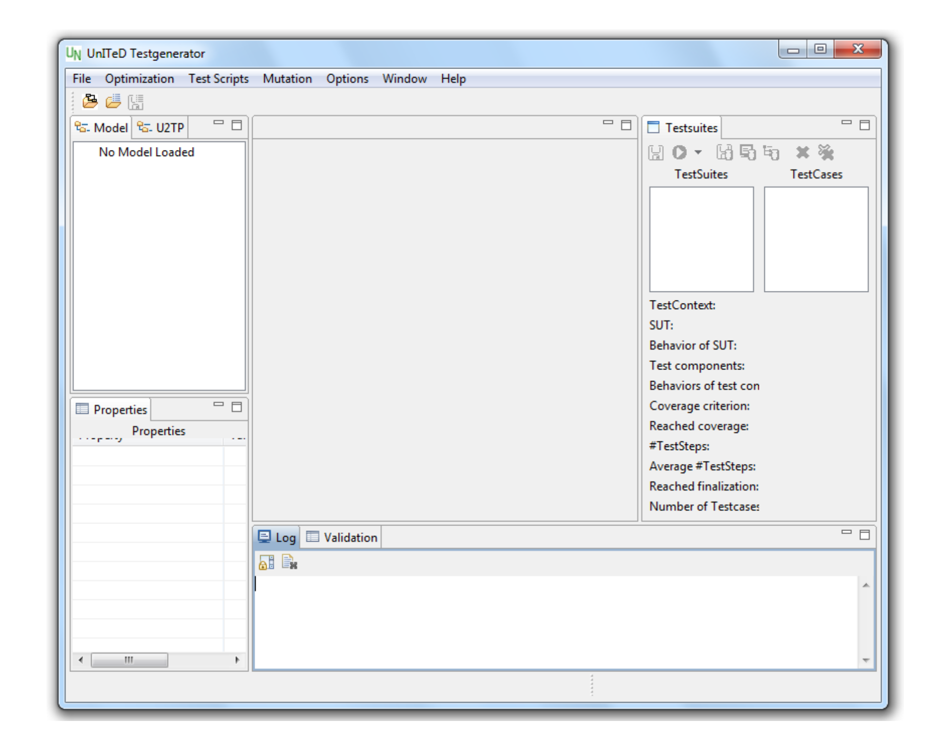


Abbildung B.1.: Startbildschirm des Werkzeugs UnITeD

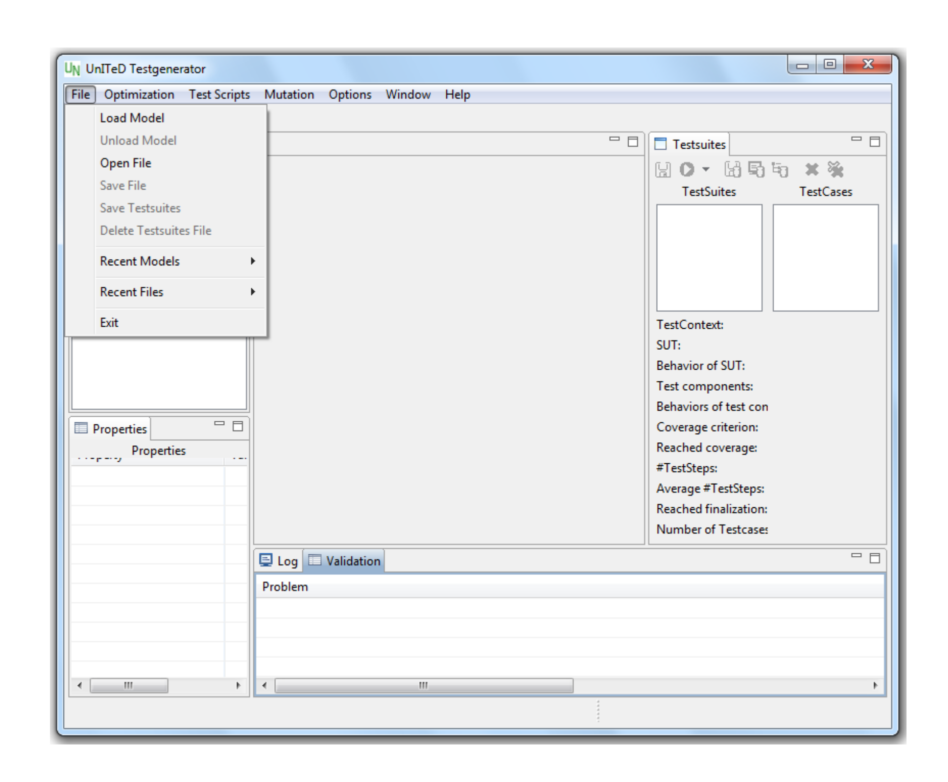


Abbildung B.2.: Menüpunkt File

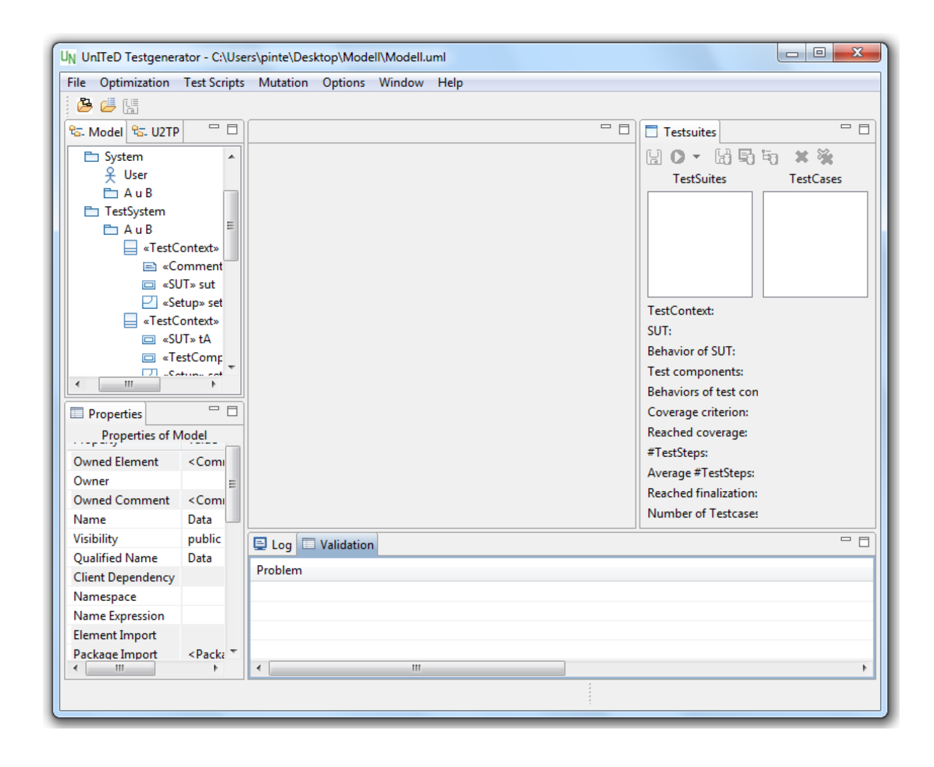


Abbildung B.3.: Anzeige des geladenen Modells im Modellbaum

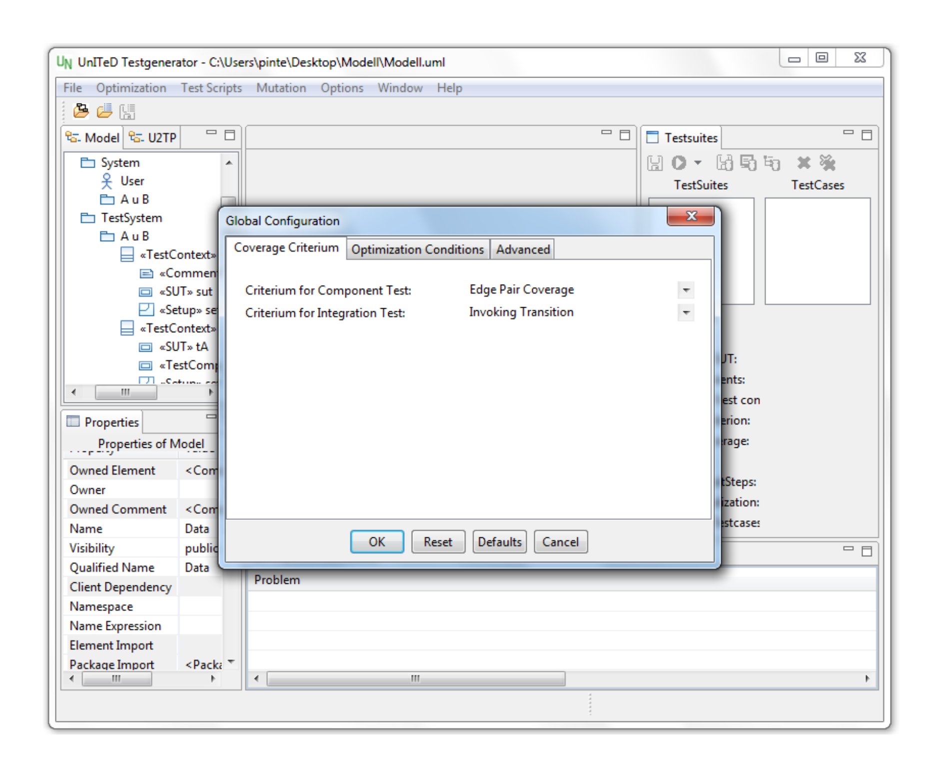


Abbildung B.4.: Konfiguration: Auswahl der Überdeckungskriterien

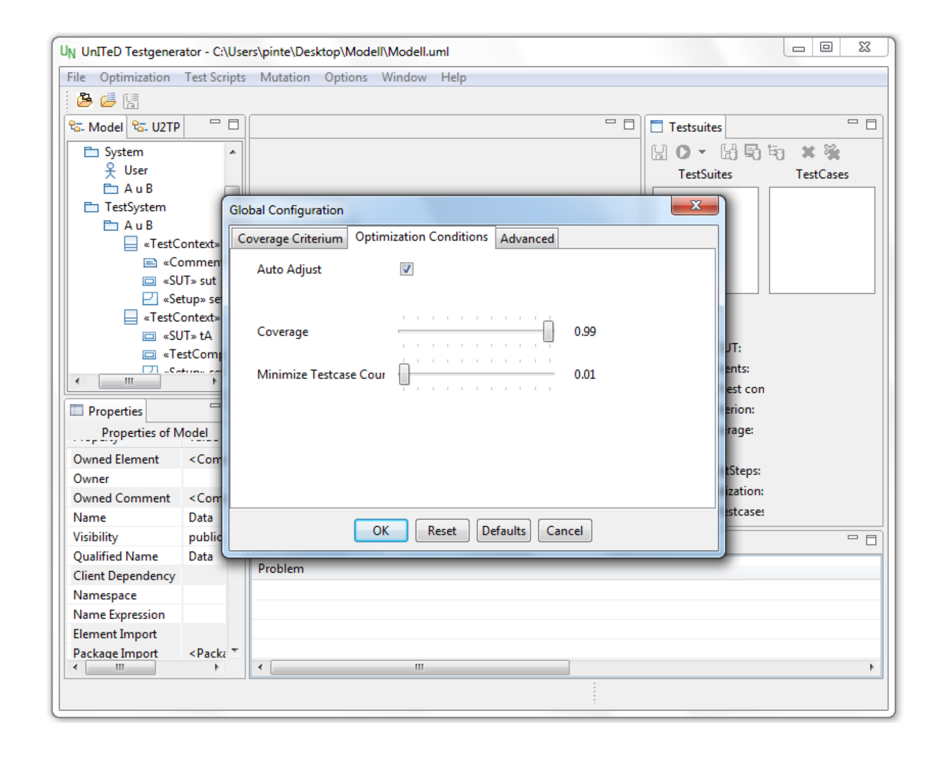


Abbildung B.5.: Konfiguration: Gewichtung der Fitnessfunktion

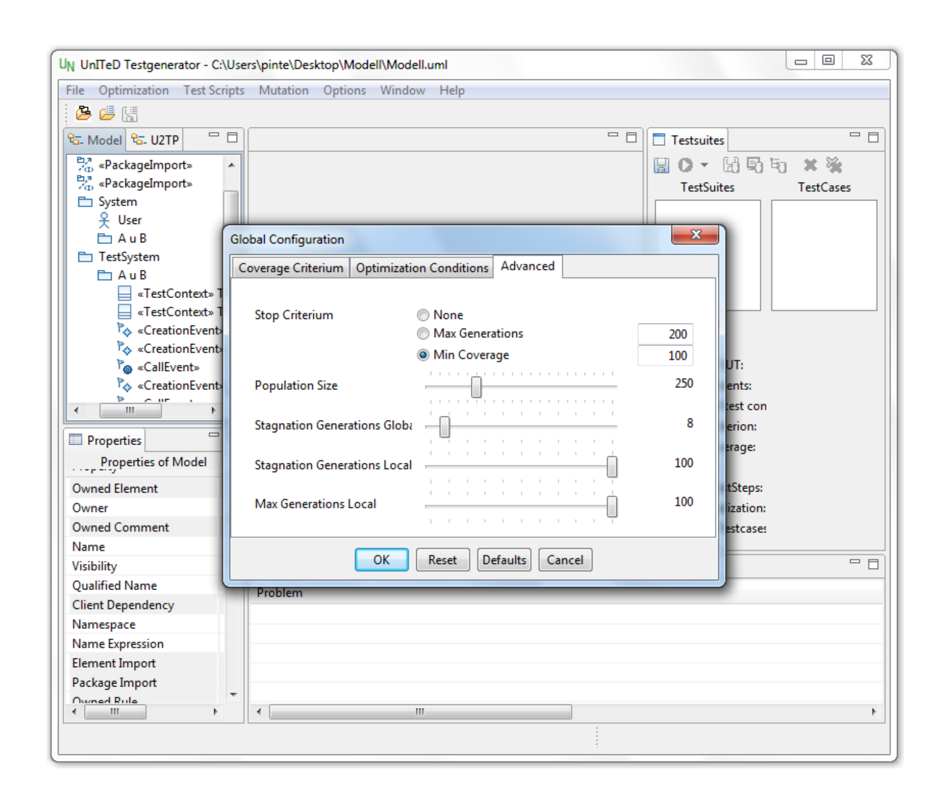


Abbildung B.6.: Konfiguration: Einstellungen für den genetischen Algortihmus

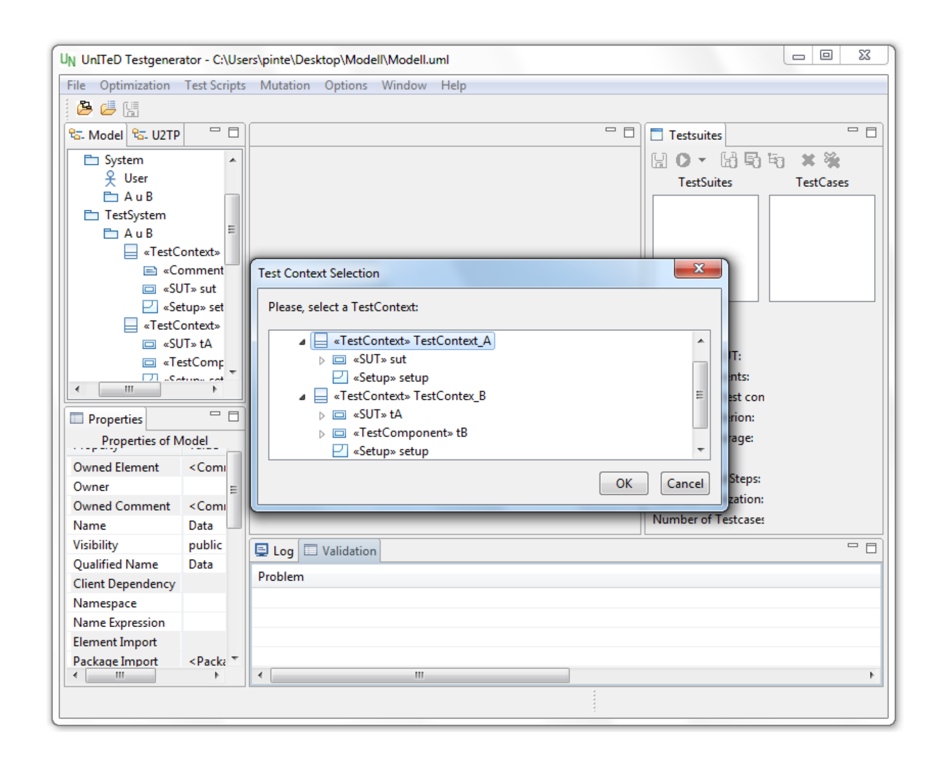


Abbildung B.7.: Auswahl TestContext

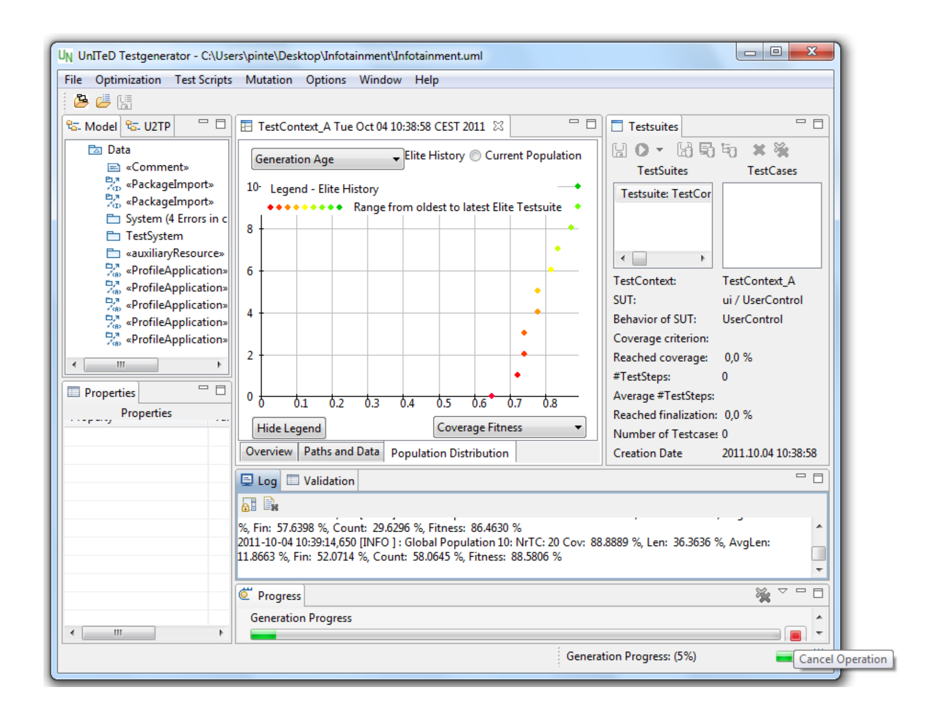


Abbildung B.8.: Anzeige Testfallgenerierungsfortschritt

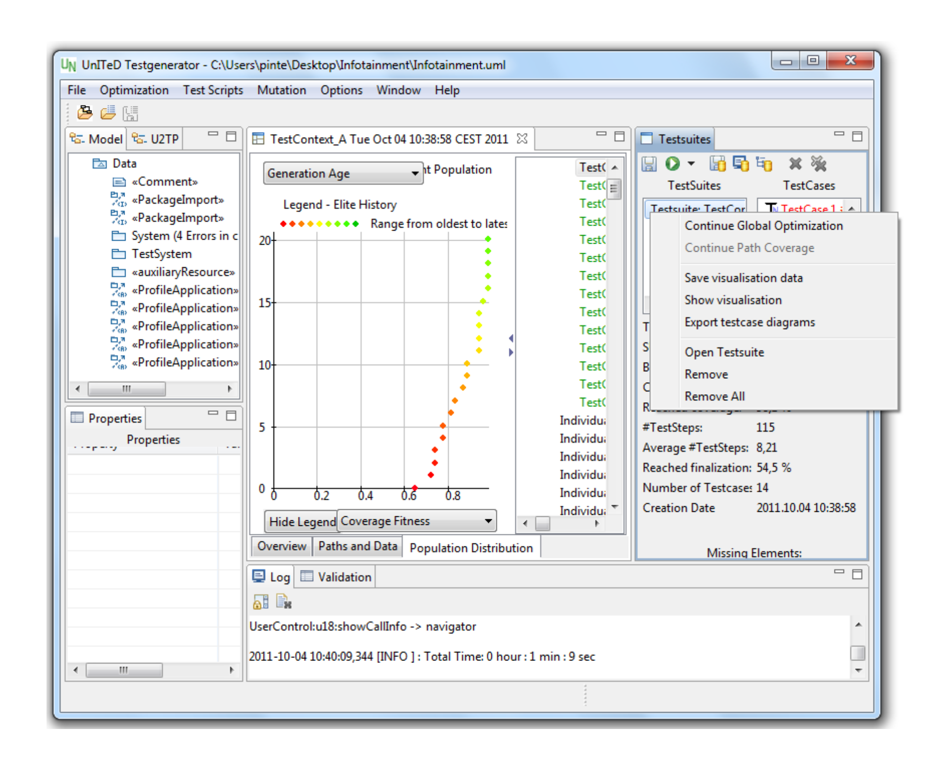


Abbildung B.9.: Kontextmenü Testsuite

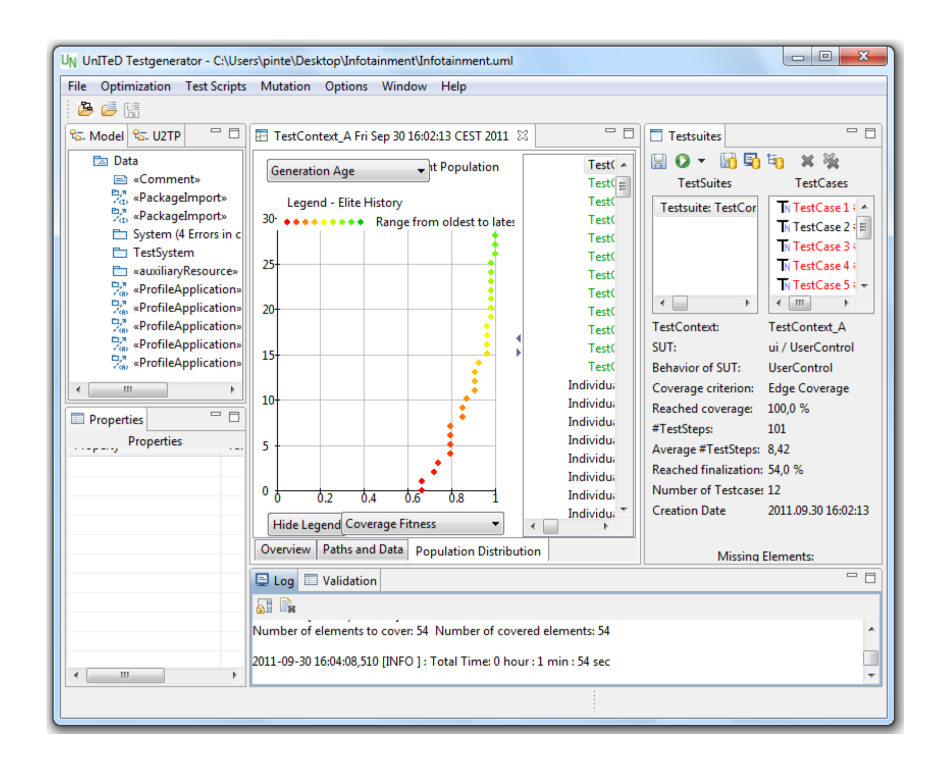


Abbildung B.10.: Anzeige der generierten Testsuite

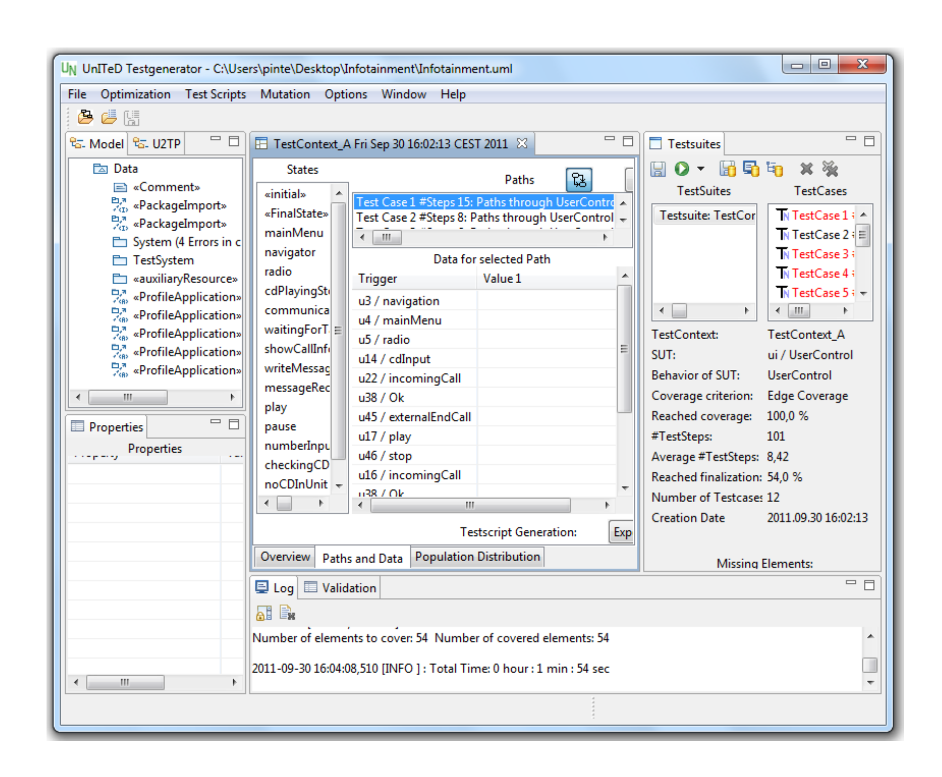


Abbildung B.11.: Anzeige einzelner Testfälle aus der generierten Testsuite

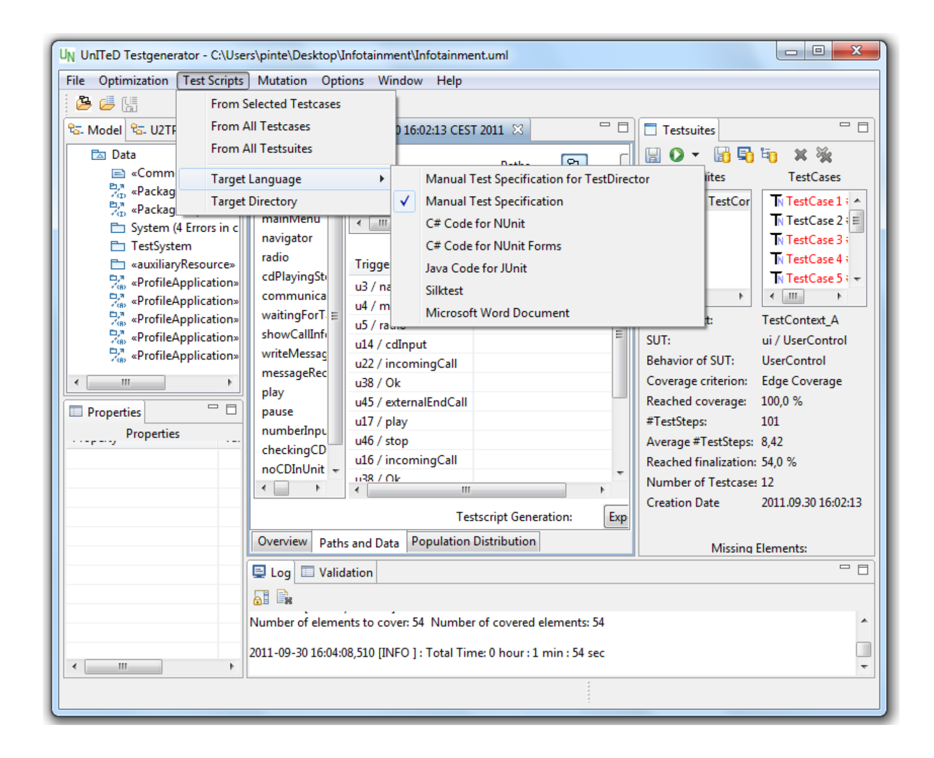


Abbildung B.12.: Generierung Testskripte

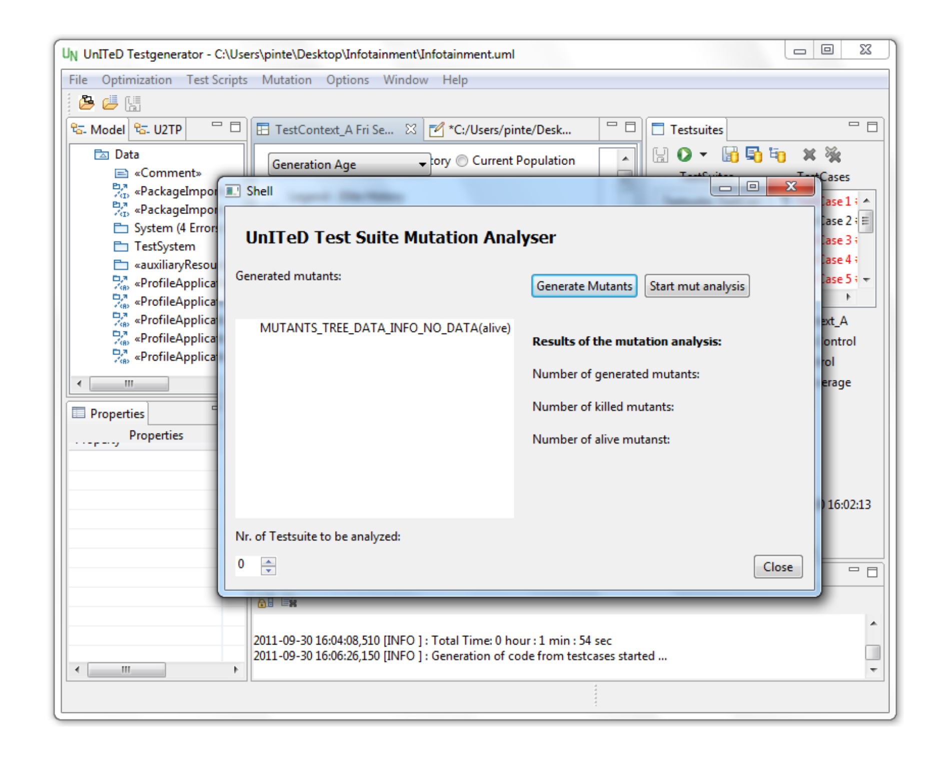


Abbildung B.13.: Mutationsanalyse

Index